Limbaje și tehnici noi de programare

Gheorghe Muscă

PROGRAMARE ÎN LIMBAJ DE ASAMBLARE

Teora

Cuprins

Capitolul 1. Introducere. Arhitectura 8086. Moduri de adresare

1.1 Introducere. Necesitatea limbajului de asamblare 1
1.2 Noțiuni introductive de hardware. Registre. Stivă 2
1.3 Reprezentări interne ale datelor 5
1.4 Tipuri de date utilizate în limbaj de asamblare 17
1.5 Moduri de adresare 25
1.6 Codificarea internă a instrucțiunilor 30

Capitolul 2. Setul de instrucțiuni 8086

2.1 Instrucțiuni de transfer 34
2.2 Instrucțiuni aritmetice și logice 44
2.3 Instrucțiuni pentru operații cu șiruri de caractere/cuvinte 68
2.4 Instrucțiuni de apel de procedură și de salt (CALL, RET, JMP) 75
2.5 Întreruperi 89
2.6 Instrucțiuni pentru controlul procesorului 94
2.7 Dezvoltarea programelor în limbaj de asamblare 95

Capitolul 3. Procesoare de 32 de biți. Coprocesoare matematice.

3.1 Arhitectura procesoarelor de 32 de biți 102
3.2 Setul de instrucțiuni al procesoarelor de 32 de biți 110
3.3 Coprocesoare matematice 114

Capitolul 4. Structura programelor. Definirea și inițializarea datelor. Operatori

4.1 Segmentare. Directive pentru definirea segmentelor 123
4.2 Definirea simplificată a segmentelor 129
4.3 Directive pentru legarea modulelor 132
4.4 Contoare de locații și directiva ORG 134
4.5 Definirea și inițializarea datelor 135
4.6 Definirea etichetelor. Directiva LABEL 137
4.7 Definirea structurilor. Operații specifice 138
4.8 Definirea înregistrărilor. Operații specifice 141
4.9 Operatori în limbaj de asamblare 143
4.10 Directive de asamblare condiționată 146

Capitolul5. Macroinstrucțiuni

5.1 Scopul macroinstrucțiunilor. Definire și expandare 149
5.2 Macroinstrucțiuni cu parametri 150

5.3 Controlul numărului de parametri actuali 155
5.4 Directive de control al listării. Etichete în macroinstrucțiuni 156
5.5 Macroinstrucțiuni repetitive 158
5.6 Operatori specifici 159
5.7 Invocare recursivă de macroinstrucțiuni 162
5.8 Definirea macroinstrucțiunilor în macroinstrucțiuni 163
5.9 Tehnici avansate de utilizare a macroinstrucțiunilor 164

Capitolul 6. Tehnici de programare în limbaj de asamblare

6.1 Decizia simplă și decizia compusă. Evaluarea condițiilor logice 168
6.2 Cicluri cu test la partea superioară și inferioară 172
6.3 Selecția. Tabele de salt sau de apel de proceduri 179
6.4 Transferul parametrilor către proceduri 182
6.5 Întoarcerea datelor de către proceduri 192
6.6 Proceduri cu zone de date proprii (variabile locale) 196
6.7 Proceduri recursive 203
6.8 Proceduri cu număr variabil de parametri 214
6.9 Tehnici avansate de programare 221
6.10 Programarea coprocesoarelor matematice 225

Capitolul 7. Interfața ASM - limbaje de nivel înalt

7.1 Transferul parametrilor. Simboluri externe 236
7.2 Dezvoltarea în mediu integrat sau prin linii de comandă 239
7.3 Modele de memorie 240
7.4 O aplicație mixtă cu tipuri de date simple 246
7.5 O aplicație mixtă cu tipuri de date structurate 252
7.6 Dezvoltarea într-un limbaj mixt (C și ASM) 259
7.7 Performanțe în ASM și C 261

Capitolul 8. Interfața cu sistemul de operare DOS

8.1 Componentele de bază ale sistemului DOS 269
8.2 Încărcarea sistemului DOS 271
8.3 Structura blocului PSP (Program Segment Prefix) 275
8.4 Structura programelor de tip COM 277
8.5 Structura și încărcarea fișierelor EXE 279
8.6 Programe de tip TSR (Terminate and Stay Resident) 282
8.7 Funcții de intrare/ieșire orientate pe handlere 299
Anexa A 310
Anexa B 315
Bibliografie 322

Capitolul 1

Introducere
Arhitectura 8086
Moduri de adresare

1.1 Introducere. Necesitatea limbajului de asamblare

Limbajul de asamblare (ASM) permite înțelegerea la nivel de amănunt a ceea ce se întâmplă în realitate într-un calculator. Familiarizarea cu un asemenea limbaj este mai mult decât benefică pentru un programator, contribuind la eficiența programelor dezvoltate, indiferent de limbajul utilizat.

Există mai multe motive pentru care programarea în ASM este necesară. Codul generat în ASM se execută în general foarte rapid. Unele module de program trebuie implementate în ASM datorită acestei viteze de lucru. Uneori, o parte a unui program este scrisă într-un limbaj de nivel înalt, iar modulele critice sunt scrise ca proceduri ASM, apelate la rândul lor de modulele de nivel înalt.

Pe de altă parte, există situații în care e nevoie de acces direct la dispozitive de intrare/ieșire sau la locații fizice de memorie, iar aceste operații nu pot fi executate în unele limbaje de nivel înalt. De exemplu, la calculatoarele personale, programele TSR și rutinele de tratare a întreruperilor sunt aproape totdeauna scrise în ASM.

Pe scurt, limbajul de asamblare oferă viteze de execuție și acces la hardware care nu pot fi disponibile (cel mai adesea) în limbajele de nivel înalt.

Un alt aspect important este cel al dezvoltării de programe pentru echipamente dedicate. Nu toate programele executabile sunt destinate calculatoarelor de uz general. Dezvoltarea masivă a microprocesoarelor a făcut ca acestea să controleze în prezent funcționarea celor mai diverse obiecte tehnice, de la mașini de gătit sau de spălat rufe până la echipamente de control industrial sau pentru comanda avioanelor. Toate aceste microprocesoare funcționează pe baza unor programe și nu trebuie să fii cine știe ce specialist ca să-ți dai seama că e puțin probabil ca un program care controlează un cuptor cu microunde sau un osciloscop să fi fost scris în PASCAL sau în BASIC. De fapt, majoritatea programelor pentru asemenea echipamente dedicate sunt scrise în limbaj de

asamblare, pentru că, într-un asemenea context, ceea ce contează este viteza de execuție și volumul foarte limitat de memorie.

Pe de altă parte, nu toate microprocesoarele sunt de uz general. Există microprocesoare specializate, destinate unor scopuri precise, cum ar fi procesoare de semnal, micro-controllere industriale, automate programabile etc. Pentru asemenea procesoare, nu se justifică (în general) dezvoltarea de compilatoare pentru limbaje de nivel înalt, programele fiind dezvoltate în limbajele specifice acestor procesoare, care sunt asemănătoare limbajelor de asamblare pentru procesoarele de uz general.

Există și un puternic rol formativ al programării în limbaj de asamblare. Un programator nu ajunge niciodată la un nivel superior (de expert), dacă nu trece (măcar o dată în viață) prin dezvoltarea de programe ASM.

Cartea de față își propune să ofere cititorului un mijloc de a-și însuși limbajul de asamblare pentru procesoarele din familia Intel (IBM-PC). Ca mediu de dezvoltare, este folosită familia de produse software Borland (Turbo Assembler, Turbo Linker și Turbo Debugger).

O atenție sporită este acordată dezvoltării de aplicații mixte (în limbaje de nivel înalt și în ASM). Ca limbaj de nivel înalt tipic, s-a ales limbajul C, pentru care există medii de dezvoltare integrate oferite de firma Borland, total compatibile cu limbajul de asamblare.

În această carte nu se insistă prea mult pe interfața software cu sistemul BIOS sau cu sistemul DOS. Există multe manuale de firmă care descriu în detaliu funcțiile și subfuncțiile BIOS și DOS. Scopul acestei cărți este de a oferi mijloace de dezvoltare de module ASM eficiente și integrarea lor în aplicații complexe.

1.2 Noțiuni introductive de hardware. Registre. Stivă

În ASM, calculatorul este văzut la nivelul hardware: adrese fizice de memorie, registre, întreruperi etc. Sunt necesare unele noțiuni pregătitoare.

Unitatea de bază a informației memorate în calculator este **bitul**. Un bit reprezintă o cifră binară (de aici și numele, care e o prescurtare de la binary digit), deci poate avea valorile 0 sau 1. Modelul hardware corespunzător este acela de **bistabil**. Un bistabil este, deci, un circuit electronic cu două stări stabile, codificate 0 și 1, capabil să memoreze un bit de informație.

Un grup de bistabili formează un **registru**. De exemplu, 8 bistabili formează un registru de 8 biți. Informația care se poate memora într-un asemenea registru poate fi codificată în binar, de la valoarea 00000000 (toți biții egali cu 0), până la valoarea 11111111 (toți biții egali cu 1). Este ușor de văzut că numărul combinațiilor care pot fi memorate este 256 (2 la puterea a 8-a). În general, un registru de n biți va putea memora 2^n combinații distincte. Aceste combinații se

numesc **octeți** sau **bytes** (dacă n = 8), respectiv **cuvinte** (dacă n = 16, 32 etc.). La procesoarele din familia Intel, cuvintele sunt de 16 sau 32 de biți (putând exista și cantități pe un număr mai mare de biți, dar toate multiplu de 8).

Memoria unui calculator este văzută ca o succesiune de octeți. Fiecare octet are asociată o adresă de memorie. Pentru a putea adresa memoria, e nevoie de un registru de adrese, a cărui lungime determină dimensiunea maximă a memoriei. Dacă avem un registru de adrese de 8 biți, atunci vom putea adresa 2^8 octeți de memorie. Procesorul 8086 are un registru de adrese de 20 de biți, deci poate adresa 2^{20} octeți de memorie (sau 1 megaoctet de memorie). Zonele de memorie vor fi reprezentate grafic pe verticală, ca succesiuni de octeți sau cuvinte, de la adrese mici către adrese mari.

O adresă cu semnificație specială este adresa instrucțiunii care se execută în mod curent. Toate procesoarele au un registru destinat acestui scop, numit **contor program** (**Program Counter**). Acest registru conține întotdeauna adresa instrucțiunii care urmează a fi executată.

O zonă specială de memorie este așa numita zonă de **stivă**. Stiva este un concept abstract de structură de date, dar procesoarele dispun de instrucțiuni mașină pentru operații cu o asemenea structură de date. O zonă de stivă este caracterizată de o adresă curentă (numită **adresa vârfului stivei**), adresată printr-un registru numit **SP** (**Stack Pointer, indicator de stivă**). Operațiile de bază cu stiva sunt:

```
PUSH (X), care se poate descrie prin actiunile:
      (SP) ← (SP) - 1
      ((SP)) ← X
POP (X), care se poate descrie prin actiunile:
      X ← ((SP))
      (SP) ← (SP) + 1
```

Notația **(SP)** înseamnă „**conținutul lui SP**", iar **((SP))**, „**conținutul locației de memorie adresate de (SP)**". Din analiza celor două operații, se vede că structura de stivă este de tip „**Primul Intrat, Primul Ieșit**" sau „**First In, First Out**", adică ultima cantitate care a fost „împinsă" în stivă printr-o operație PUSH va fi cea care se va extrage la următoarea operație POP. La procesorul 8086, cantitățile transferate în stivă sunt cuvinte de 16 biți, deci adresa curentă a vârfului stivei (conținută în SP) este adunată sau scăzută cu 2.

Stiva poate fi folosită explicit pentru salvări și refaceri de date. Ea este folosită implicit în mecanismul de apel de procedură. Când se apelează o procedură cu numele PR, printr-un apel generic de forma **CALL PR**, se va da controlul (adică se va produce un salt) la prima instrucțiune a procedurii. Acest salt nu înseamnă

altceva decât că în registrul contor program se va depune adresa primei instrucțiuni din procedură.

Se pune însă problema revenirii din procedură, la execuția unei instrucțiuni generice **RETURN**. Controlul trebuie să revină în programul apelant și să se execute instrucțiunea de după apelul de procedură.

Acest fapt este posibil, deoarece la execuția instrucțiunii **CALL**, înainte de a se executa saltul la procedură, se salvează în stivă conținutul contorului program (adică adresa instrucțiunii imediat următoare apelului de procedură). Instrucțiunea **RETURN** nu face altceva decât să extragă această adresă din stivă și să o plaseze în contorul program, execuția continuând cu instrucțiunea de după apelul de procedură. Acest mecanism este ilustrat în Figura 1.1.

Un program ASM (ca de altfel și unul în limbaj de nivel înalt) este organizat în trei spații de memorie:

- spațiul (zona) de cod - acesta cuprinde instrucțiunile care compun programul sau codul;
- spațiul (zona) de date - acesta cuprinde datele statice definite în program (care se găsesc la adrese fixe);
- spațiul (zona) de stivă - acesta cuprinde stiva rezervată programului.

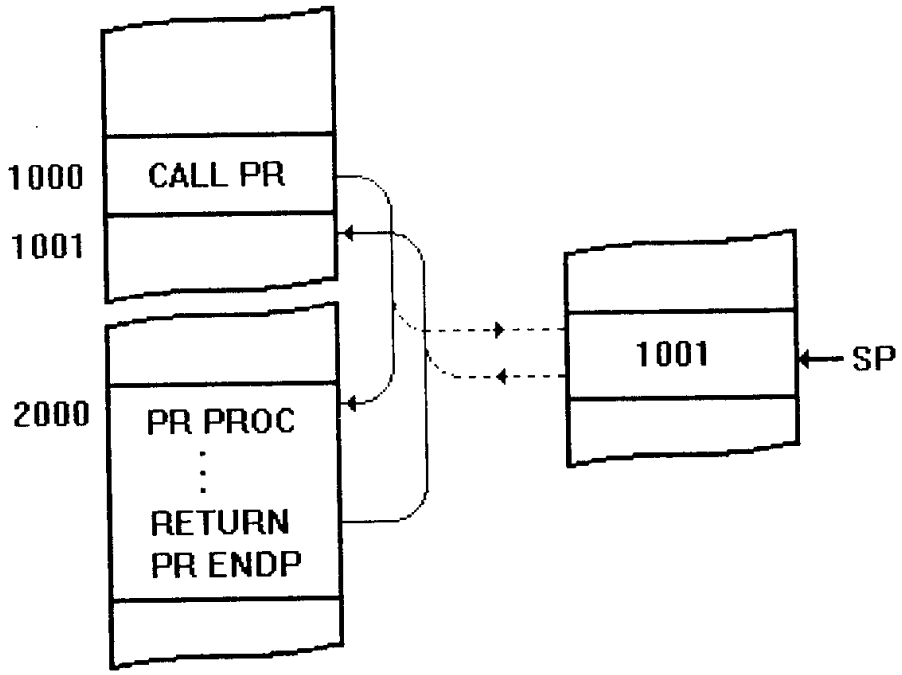


Figura 1.1 Rolul stivei la apelul unei proceduri

Zona de date poate fi detaliată în date constante, date inițializate sau neinițializate etc.

Trebuie remarcat că orice program executabil are structura de mai sus, indiferent de limbajul în care a fost scris, dar în ASM această structură este vizibilă și gestionată de către programator. Acesta poate chiar schimba semnificațiile zonelor respective. De exemplu, există situații (programe TSR) în care datele sunt definite în același spațiu de memorie cu instrucțiunile.

Pe lângă cele trei spații de memorie, mai există și așa numitul spațiu de memorie disponibilă. Acesta folosește la alocări dinamice de memorie (la momentul execuției programului) și este gestionat în mod indirect, prin apeluri către sistemul de operare.

1.3 Reprezentări interne ale datelor

1.3.1 Baze de numerație

În tehnica de calcul, pe lângă baza de numerație 10, sunt folosite bazele de numerație 2, 8 și 16, iar sistemele de numerație respective se numesc binar, octal și hexazecimal. În sistemul hexazecimal, cifrele de la 10 la 15 se notează cu literele de la A până la F. Tabelul 1.1 ilustrează corespondența dintre sistemele zecimal, binar, octal și hexazecimal. Trebuie reținut faptul esențial că, în memoria calculatorului, informația de orice fel (date sau programe) este întotdeauna reprezentată în formă binară, deci ca secvențe de cifre 0 sau 1.

Deoarece este greu de operat cu numere mari în baza 2, pentru exprimarea unor cantități binare se folosesc bazele 8 și 16. De exemplu, numărul 255 se poate exprima în aceste baze prin $255_{(10)} = 11111111_{(2)} = FF_{(16)} = 377_{(8)}$.

Informația este organizată în calculator pe grupe de câte 8 cifre binare (biți). Un asemenea grup se mai numește și octet (sau byte). Octetul este unitatea de măsură în care se exprimă volumul memoriei unui calculator.

Deoarece dimensiunea memoriei este totdeauna o putere a lui 2, multiplii folositi pentru octeți nu sunt 100, 1000 etc., ci puteri adecvate ale lui 2. Astfel, un număr de 2^{10} = 1024 de octeți se mai numește și kilooctet sau kilobyte (pe scurt KO sau KB). Similar, un număr de 2^{20} = 10264576 octeți se numește megaoctet sau megabyte (MO sau MB). Se mai folosește și gigaoctetul (gigabyte-ul), care este egal cu 230 octeți. Așadar, multiplii folosiți în tehnica de calcul sunt:

- kilo = 210 = 1024
- mega = 220 = 10264576
- giga = 230 = 1073741824

Zecimal	Binar	Octal	Hexazecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Tabelul 1.1 Reprezentarea numerelor în bazele 10, 2, 8 și 16

1.3.2 Reprezentarea numerelor întregi

Numerele întregi pot fi reprezentate pe unul sau mai mulți octeți, deci pe un număr n de biți. Deoarece fiecare bit poate lua două valori (0 și 1), numărul total de valori distincte este 2^n.

În cazul numerelor fără semn, valoarea internă a biților (octeților) reprezintă chiar valoarea numărului. Este ușor de văzut că, dacă reprezentarea este pe n biți, domeniul posibil de valori este de la 0 la 2^n-1. Pentru n = 3 rezultă domeniul din Tabelul 1.2.

În ceea ce privește reprezentarea numerelor cu semn, trebuie spus că valorile interne care se pot reprezenta (deci configurațiile de biți) sunt aceleași ca la numerele fără semn. Aceste configurații variază între 0000...0000 (toți biții egali cu 0) și 1111...1111 (toți biții egali cu 1).

Se pune deci problema ca, printr-o convenție adecvată, să se considere o parte din aceste configurații de biți ca reprezentând numere întregi pozitive, iar cealaltă parte numere negative.

Reprezentare internă	Valoare
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Tabelul 1.2 Reprezentarea numerelor fără semn pe 3 biți

Există mai multe sisteme de reprezentare, dintre care cel mai răspândit este sistemul de reprezentare în **complement față de 2**. În acest sistem, bitul cel mai semnificativ joacă un rol special, anume de a preciza semnul numărului (din acest motiv se numește bit de semn). Dacă bitul de semn este 0, numărul reprezentat este pozitiv, iar dacă bitul de semn este 1, numărul este negativ.

În ceea ce privește valoarea absolută (modulul) a numărului, ea se obține diferențiat, funcție de semnul numărului, în felul următor:

- dacă bitul de semn este 0, atunci configurația internă reprezintă chiar valoarea numărului;
- dacă bitul de semn este 1, atunci valoarea absolută a numărului se obține prin complement față de 2, adică prin complementarea tuturor biților (0 devine 1 și reciproc) și prin adunarea apoi a valorii 1.

Această regulă se aplică pentru ambele sensuri de conversie (de la reprezentări interne la numere cu semn și reciproc).

De exemplu, fie numărul de biți n = 3 și să considerăm reprezentarea 111. Bitul de semn este 1, deci este vorba de un număr negativ. Prin complementare se obține 000, iar prin adunare cu 1 se obține 001, adică valoarea absolută 1. Astfel, reprezentarea binară 111 pe 3 biți corespunde numărului -1.

Reprezentarea în complement față de 2 pe n biți are următoarele proprietăți generale:

- domeniul de reprezentare este -2^{n-1} ... $2^{n-1}-1$;
- există o singură reprezentare pentru 0, anume 000...000;
- bitul cel mai semnificativ este bit de semn;
- calculul valorilor absolute pentru numere negative se face prin aplicarea regulii complementului față de 2.

În Tabelul 1.3 sunt ilustrate domeniile de valori pentru numere cu semn pe n = 3 biți.

Să remarcăm faptul important că o aceeași reprezentare internă poate avea semnificații (interpretări) diferite, după cum se consideră reprezentarea cu semn sau fără semn. De exemplu, valoarea internă 1010 este interpretată ca -6 în reprezentarea cu semn și ca +10 în reprezentarea fără semn pe 4 biți. Se vede că diferența dintre aceste două valori posibile este chiar 2^n (în cazul de față 16).

De aici se poate deduce o altă regulă pentru aflarea rapidă a reprezentării numerelor negative: se adună 2^n, iar valoarea obținută se reprezintă ca număr fără semn.

De exemplu, ca să obținem reprezentarea lui -120 pe 8 biți, adunăm 256 și obținem valoarea pozitivă 136. Reprezentarea lui 136 în baza 2 va fi atunci reprezentarea în complement față de 2 a lui -120, adică 10001000 sau 88H.

Reprezentare internă	Valoare
000	0
001	1
010	2
011	3
100	-4
101	-3
110	-2
111	-1

Tabelul 1.3 Reprezentarea numerelor cu semn pe 3 biți

În diverse sisteme de afișare a datelor numerice, în care este folosită numai baza 10, se utilizează codificarea de tip **BCD** care înseamnă „**Zecimal Codificat în Binar**" (în engleză, „**Binary Coded Decimal**").

În acest sistem, se reprezintă o cifră zecimală pe un grup de 4 biți (4 biți pot codifica 16 valori distincte, deci 6 valori nu vor fi folosite). Un octet va codifica deci 2 cifre zecimale. De exemplu, octetul cu valoarea 01011001 reprezintă codificarea BCD pentru numărul zecimal 59 (primul grup de 4 biți reprezintă cifra 5, iar al doilea grup cifra 9). Dacă este necesar, se pot considera n octeți pentru memorarea a 2n cifre zecimale.

Calculatoarele dispun, de obicei, de instrucțiuni speciale pentru calcule în format BCD. Sunt unele probleme care trebuie avute în vedere, și anume operațiile de

corecție care trebuie făcute după (sau înaintea) operațiilor în BCD, deoarece nu toate cele 16 combinații binare posibile sunt cifre BCD corecte.

Formatul BCD se mai numește și **BCD împachetat**, pentru a-l deosebi de formatul **BCD despachetat**, în care se reprezintă o cifră BCD pe un octet. În terminologia firmei INTEL (reflectată în denumirile instrucțiunilor procesoarelor), formatul BCD despachetat este denumit (oarecum incorect) format ASCII. Justificarea denumirii este că, prin adăugarea valorii 30H (codul ASCII al cifrei 0), se obține codul ASCII al cifrei reprezentate de octetul respectiv.

Toate cele trei sisteme de reprezentare a numerelor întregi ilustrează faptul că datele din memoria calculatorului se pot interpreta diferit. Aceste interpretări sunt făcute de programul care folosește datele. De exemplu octetul 10010100 poate reprezenta numărul zecimal pozitiv 148, numărul zecimal negativ -108 sau numărul zecimal pozitiv 94, după cum se lucrează cu sistemul de reprezentare fără semn, cu semn sau, respectiv, cu sistemul BCD.

1.3.3 Reprezentarea numerelor reale

Numerele reale (fracționare) se pot reprezenta în două moduri distincte, care sunt numite reprezentare în virgulă fixă și reprezentare în virgulă mobilă.

Reprezentarea în virgulă fixă

În reprezentarea în virgulă fixă, se consideră un număr finit de cifre semnificative, atât pentru partea întreagă a numărului, cât și pentru cea fracționară. Considerând baza de numerație b, n cifre pentru partea întreagă și m cifre pentru partea fracționară, un număr fracționar x se reprezintă prin expresia:

$$x = i_n \cdot b^n + ... + i_1 \cdot b^1 + i_0 \cdot b^0 + f_1 \cdot b^{-1} + ... + f_m \cdot b^{-m}$$

în care cifrele $i_n...i_0$ reprezintă partea întreagă, iar cifrele $f_1...f_m$ reprezintă partea fracționară a numărului x. Cifrele sunt considerate în baza b. Un asemenea număr se scrie pozițional în forma:

$$i_n...i_1 i_0 . f_1...f_m \ _{(b)}$$

(în tehnica de calcul, virgula se notează cu un punct). Exemple de astfel de numere fracționare pot fi:

$$123.25_{(10)} \quad 1AB34.FF52_{(16)} \quad 1000110.11011_{(2)}$$

Într-o reprezentare internă concretă (pe un sistem de calcul), baza b și numărul de cifre de la stânga și de la dreapta virgulei (n și m) sunt fixate aprioric. De exemplu, lucrând cu baza 10 și presupunând că o precizie de 0.001 este suficientă, se vor memora doar 3 cifre după virgulă. Similar, dacă se presupune că numerele care trebuie reprezentate nu depășesc valoarea 10000, se vor

aloca 4 cifre la stânga virgulei. Se vor reprezenta astfel numere zecimale în domeniul 0.001 ... 9999.999, cu o precizie de 0.001.

Cifrele respective vor fi numite mii, sute, zeci, unități, zecimi, sutimi, miimi etc. și se vor reprezenta fie pe câte un octet, fie pe câte 4 biți. Memorarea semnului unui astfel de număr se face într-un câmp separat. Tabelul 1.4 exemplifică reprezentarea în virgulă fixă a numerelor 1.130, -1230.192 și 10.000.

Semn	Mii	Sute	Zeci	Unități	Zecimi	Sutimi	Miimi
+	0	0	0	1	0	1	3
-	1	2	3	0	1	9	2
+	0	0	1	0	0	0	0

Tabelul 1.4 Reprezentarea în virgulă fixă (b = 10, n = 4, m = 3)

Se observă că punctul zecimal nu se reprezintă, deoarece poziția sa este cunoscută (este fixă, între coloanele unităților și ale zecimilor). De altfel, numele metodei de reprezentare provine de la această proprietate.

Uneori, se reprezintă numărul real respectiv înmulțit cu 10 la o putere egală cu numărul de cifre de după virgulă. De exemplu, numărul 1.130 se poate reprezenta intern ca numărul întreg 1130, iar numărul -1230.192 ca numărul întreg -1230192 (în complement față de 2, pentru a memora și semnul). În operațiile aritmetice care se vor face, se va ține seama de faptul că numerele sunt înmulțite cu 1000 și se va reprezenta punctul zecimal după a treia cifră de la dreapta la stânga.

Trebuie remarcat că, într-o astfel de reprezentare cu număr fix de câmpuri, nu se poate reprezenta decât un subset finit al numerelor reale, în exemplul de mai sus acest subset fiind cuprins între -9999.999 și +9999.999, cu pasul 0.001. Trebuie reținut, de asemenea, că toate valorile din subsetul respectiv sunt reprezentate exact.

De asemenea, este evident faptul că operațiile aritmetice de înmulțire și împărțire se fac într-un mod aproximativ. Înmulțind 1.121 cu 2.250 se obține valoarea teoretică 2.52225, dar aceasta nu se poate reprezenta în sistemul cu 3 cifre după punctul zecimal. Ca atare, rezultatul este rotunjit la valoarea cea mai apropiată care se poate reprezenta, anume 2.522. Operațiile de adunare și scădere se fac exact, cu condiția ca rezultatul să nu depășească limitele domeniului de reprezentare.

În concluzie, reprezentarea în virgulă fixă este caracterizată de numărul de cifre reprezentate și de numărul de cifre de după virgulă (poziția virgulei).

Reprezentarea în virgulă fixă se folosește în unele sisteme de conducere cu calculator a mașinilor-unelte industriale (sisteme de poziționare) și - foarte important - în sistemele de programe financiare și contabile.

La acestea din urmă, reprezentarea în virgulă fixă are avantajul că toate puterile lui 10 (pozitive și negative) care fac parte din domeniul de valori sunt reprezentate exact.

De exemplu, pentru a reprezenta sume de bani, este suficient să avem două cifre după virgulă. La un bilanț financiar, când se adună sau se scad sume de bani, este esențial ca operațiile de adunare și scădere să se facă exact.

Din motivele prezentate, în programele aplicative pentru finanțe (în general pentru domeniul economic) se folosește sistemul de reprezentare în virgulă fixă.

Reprezentarea în virgulă mobilă

Reprezentarea în virgulă mobilă se folosește cu precădere în domeniile științifice și tehnice (sau, altfel spus, în toate domeniile în afară de cele economico-financiare).

În această reprezentare, numerele sunt considerate de forma:

$$(-1)^S \cdot M \cdot B^E$$

unde:

- S este numit bit de semn și este 1 pentru numere negative și 0 pentru numere pozitive;
- M este numită mantisă (sau fracție) și este un număr pozitiv subunitar reprezentat în baza B;
- B este numită bază (uzual este 2 sau 16);
- E este numit exponent și este un număr întreg cu semn.

Mantisa M se numește **normalizată** dacă prima cifră după virgulă este cifră semnificativă (este diferită de zero).

Majoritatea sistemelor actuale folosesc baza B = 2. În acest caz, faptul că prima cifră a mantisei este diferită de 0 înseamnă că aceasta este obligatoriu 1. Astfel, mantisa M verifică condiția (scrisă în baza 2) :

$$0.1_{(2)} \leq M < 1$$

sau, în baza 10:

$$0.5_{(10)} \leq M < 1$$

Exponentul se reprezintă de obicei deplasat, în sensul că se memorează un număr de forma E + K, unde K este o constantă astfel aleasă încât E + K să fie totdeauna pozitiv. În felul acesta se rezolvă problema memorării semnului exponentului. De cele mai multe ori, deoarece prima cifră semnificativă a

mantisei M este 1, această valoare nu se mai reprezintă, câștigându-se astfel un bit în spațiul de memorare (în care se poate memora bitul de semn S).

Cea mai des folosită reprezentare standardizată este cea numită în **simplă precizie**, descrisă în cele ce urmează. În acest sistem, mantisa M are 24 de biți, iar exponentul E are 8 biți și se reprezintă intern deplasat (adunat) cu 127. Mantisa este normalizată și înmulțită cu 2, astfel încât condiția de normalizare este $1 \leq M < 2$.

Bitul cel mai semnificativ al mantisei nu se mai reprezintă intern, deoarece este totdeauna 1. Astfel, un număr real în simplă precizie se reprezintă pe 32 de biți (4 octeți).

Notând cu f mantisa (fracția) care se reprezintă intern, cu e exponentul deplasat și cu s bitul de semn, valoarea unui număr real este dată de relația:

$$(-1)^s \cdot 1.f_{22}f_{21}...f_1f_0 \cdot 2^{e-127}$$

în care s este bitul de semn, $e = e_7e_6...e_1e_0$ este exponentul deplasat cu 127, iar $f = f_{22}f_{21}...f_1f_0$ este mantisa (fracția) normalizată. Bitul cu valoarea 1 din formula de mai sus (bitul 23 al mantisei) nu se reprezintă intern. Cei patru octeți sunt reprezentați în Tabelul 1.5, în ordinea așezării lor în memorie (în sens crescător al adreselor de memorie).

Nu toate combinațiile posibile de biți din tabel sunt reprezentări valide. Astfel, numărul 0.0 se reprezintă în mod unic prin s = 0, f = 0 și e = 0, aceasta fiind singura valoare pentru care e sau f pot fi nule. În afara valorii reale 0.0, exponentul deplasat trebuie să fie cuprins între 1 și 254.

Exponentul 0 este permis numai pentru reprezentarea valorii reale 0.0 (deci și cu f = 0 și s = 0). Exponentul 255 (0FFH) este folosit pentru reprezentarea unor valori de excepție. Astfel, reprezentările cu e = 0FFH, f = 7FFFFFH (toți biții 1) și s = 0 sau 1 sunt folosite pentru valorile notate **+INF** și **-INF**, care sunt cazuri de excepție (depășiri aritmetice). Orice alte valori cu exponent 0 sau 255 sunt incorecte. Aceste valori incorecte mai sunt numite și **NAN** (de la **Not A Number**). Funcție de bitul de semn, pot exista valori +NAN sau -NAN.

f7	f6	f5	f4	f3	f2	f1	f0
f15	f14	f13	f12	f11	f10	f9	f8
e0	f22	f21	f20	f19	f18	f17	f16
s	e7	e6	e5	e4	e3	e2	e1

Tabelul 1.5 Reprezentarea în virgulă mobilă în simplă precizie

Alte valori reprezentative sunt:

- cel mai mic număr pozitiv reprezentabil este caracterizat de s = 0, f = 0, e = 1 și are valoarea aproximativă $1.17 \cdot 10^{-38}$;
- cel mai mare număr pozitiv reprezentabil este caracterizat de s = 0, f = 7FFFFF, e = 0FEH și are valoarea aproximativă $3.4 \cdot 10^{38}$.

Să considerăm câteva exemple de reprezentare.

- Numărul real 0.0 se reprezintă prin 4 octeți nuli.
- Numărul real 1 se exprimă prin semnul s = 0, mantisa 1.0 și exponentul 0 ($1 = (-1)^0 \cdot 1.0 \cdot 2^0$). Fracția internă este f = 0, iar exponentul deplasat este e = 127 (sau e = 7FH).
- Numărul real 2 se exprimă prin semnul s = 0, mantisa 1.0 și exponentul 1 ($2 = (-1)^0 \cdot 1.0 \cdot 2^1$). Fracția internă este f = 0, iar exponentul deplasat este e = 128 (sau e = 80H).
- Numărul real 0.5 se exprimă prin semnul 0, mantisa 1.0 și exponentul -1 ($0.5 = (-1)^0 \cdot 1.0 \cdot 2^{-1}$). Fracția internă este f = 0, iar exponentul deplasat este e = 126 (sau e = 7EH).
- Numărul real -1 se exprimă prin semnul 1, mantisa 1.0 și exponentul 0 ($-1 = (-1)^1 \cdot 1.0 \cdot 2^0$). Fracția internă este f = 0, iar exponentul deplasat este e = 127 (sau e = 7FH).

Scriind bitul s, biții fracției f și ai exponentului deplasat e ca în Tabelul 2.4, se obțin reprezentările pe 4 octeți de mai jos:

- 0.0 = 00 00 00 00 (s = 0, f = 0, e = 0)
- 1.0 = 00 00 80 3F (s = 0, f = 0, e = 7FH)
- 2.0 = 00 00 00 40 (s = 0, f = 0, e = 80H)
- 0.5 = 00 00 00 3F (s = 0, f = 0, e = 7EH)
- -1.0 = 00 00 80 BF (s = 1, f = 0, e = 7FH)

Reprezentarea în virgulă mobilă este caracterizată prin numărul de cifre al mantisei (fracției) și numărul de cifre al exponentului.

Să observăm că, la aceeași mantisă, putem avea exponenți diferiți ceea ce înseamnă că virgula nu mai este pe o poziție fixă, ci se deplasează după cât de mare este exponentul. Acest fapt a condus la numele reprezentării (virgulă mobilă). Avantajul acestei reprezentări este domeniul foarte mare. În simplă precizie putem reprezenta numere cuprinse aproximativ între -10^{38} și 10^{38}.

Trebuie, de asemenea, remarcat că se reprezintă corect numai numerele care se pot exprima ca sume de puteri (pozitive sau negative) ale lui 2 (în general ale bazei B). Exemplele de mai sus sunt din această categorie. Numerele 0.1, 0.01, 0.3 nu se reprezintă exact în virgulă mobilă cu baza 2. De aceea, acest sistem nu se folosește în programe pentru domeniile economico-financiare.

Calculele aritmetice sunt aproximative, dar precizia este de cele mai multe ori suficientă pentru aplicațiile uzuale. În reprezentarea în simplă precizie (pe 4 octeți), precizia asigurată este de 10^{-7}.

Reprezentarea în precizie dublă (8 octeți) este caracterizată de o mantisă normalizată pe 53 de biți și un exponent pe 11 biți (deplasat cu 1023). Valoarea unui număr real în dublă precizie este dată de expresia:

$$(-1)^s \cdot 1.\, f_{51}f_{50}...f_1f_0 \cdot 2^{e-1023}$$

în care s este bitul de semn, $e = e_{10}e_9...e_1e_0$ este exponentul deplasat cu 1023, iar $f = f_{51}f_{50}...f_1f_0$ este mantisa (fracția) normalizată. Bitul cu valoarea 1 din formula de mai sus (bitul 52 al mantisei) nu se reprezintă intern. Structura celor 8 octeți (în ordinea crescătoare a adreselor) este cea din Tabelul 1.6 .

Valorile corecte pentru exponentul deplasat sunt cuprinse între 1 și 1022. Numărul 0.0 se prezintă în mod unic cu s = f = e = 0, iar valorile cu exponent 1023 sunt folosite pentru +INF și -INF.

f7	f6	f5	f4	f3	f2	f1	f0
f15	f14	f13	f12	f11	f10	f9	f8
f23	f22	f21	f20	f19	f18	f17	f16
f31	f30	f29	f28	f27	f26	f25	f24
f39	f38	f37	f36	f35	f34	f33	f32
f47	f46	f45	f44	f43	f42	f41	f40
e3	e2	e1	e0	f51	f50	f49	f48
s	e10	e9	e8	e7	e6	e5	e4

Tabelul 1.6 Reprezentarea în virgulă mobilă în dublă precizie

Reprezentarea în precizie extinsă (10 octeți) este caracterizată de o mantisă normalizată pe 64 de biți și un exponent pe 15 biți (deplasat cu 16383). Valoarea unui număr real în precizie extinsă este dată de expresia:

$$(-1)^s \cdot 1.\, f_{62}f_{61}...f_1f_0 \cdot 2^{e-16383}$$

în care s este bitul de semn, $e = e_{14}e_{13}...e_1e_0$ este exponentul deplasat cu 16383, iar $f = f_{62}f_{61}...f_1f_0$ este mantisa (fracția) normalizată. Bitul cu valoarea 1 din formula de mai sus (bitul 63 al mantisei) se reprezintă intern ca un bit totdeauna egal cu 1, cu excepția valorii zero, când acest bit este 0. Structura celor zece octeți (în ordinea crescătoare a adreselor) este cea din Tabelul 1.7.

f7	f6	f5	f4	f3	f2	f1	f0
f15	f14	f13	f12	f11	f10	f9	f8
f23	f22	f21	f20	f19	f18	f17	f16
f31	f30	f29	f28	f27	f26	f25	f24
f39	f38	f37	f36	f35	f34	f334	f32
f47	f46	f45	f44	f43	f42	f41	f40
f55	f54	f53	f52	f51	f50	f49	f48
1	f62	f61	f60	f59	f58	f57	f56
e7	e6	e5	e4	e3	e2	e1	e0
s	e14	e13	e12	e11	e10	e9	e8

Tabelul 1.7 Reprezentarea în virgulă mobilă în precizie extinsă

Microprocesoarele din familia 80x86 sunt dotate cu circuite specializate (coprocesoare matematice) care folosesc cele trei tipuri de reprezentări în virgulă mobilă de mai sus. De asemenea, limbajul de asamblare specific acestor procesoare dispune de directive pentru definirea de constante reale în toate cele trei precizii.

1.3.4 Reprezentarea datelor nenumerice

Pe lângă date numerice, în memoria calculatorului trebuie reprezentate și alte tipuri de date. Un exemplu important îl reprezintă caracterele alfabetice (litere mari și mici), caracterele numerice (cifre zecimale), semnele de punctuație și așa-numitele caractere de control (folosite pentru comanda diverselor echipamente periferice).

Deși există mai multe sisteme de codificare a acestor tipuri de date, pe majoritatea calculatoarelor actuale (inclusiv pe IBM-PC) se folosește codul standard **ASCII**. Denumirea sa provine din inițialele de la **American Standard Code for Information Interchange** (**Codul Standard American pentru Schimb de Informații**).

Codul ASCII standard este un cod pe 7 biți, deci cuprinde 128 de caractere distincte. Un caracter ASCII se reprezintă pe un octet, în care bitul cel mai semnificativ este 0. Domeniul de valori este deci de la 0 la 127 (în zecimal) sau de la 00 la 7F (în hexazecimal).

Dintre cele 128 de caractere, 32 sunt caractere de control și nu au reprezentări grafice (nu sunt afișabile). Restul de 96 de caractere pot fi afișate pe ecranul calculatorului, la imprimantă etc. Caracterele afișabile se notează de obicei prin

scrierea simbolului grafic respectiv între apostrofuri. De exemplu, 'A' înseamnă codul ASCII corespunzător literei A mare, '!' înseamnă codul ASCII pentru semn de exclamare etc.

Calculatorul IBM-PC lucrează cu un așa-numit cod ASCII extins, în care se folosesc toți cei 8 biți ai unui octet. Setul ASCII standard este un subset al acestui cod ASCII extins.

Tabelul 1.8 cuprinde toate codurile ASCII. Coloanele corespund primei cifre hexa, iar liniile, celei de-a doua cifre. Primele două coloane (în afară de codul 20H) și ultimul cod (DEL sau 7F) reprezintă cele 32 de caractere de control.

Deoarece caracterele de control nu au reprezentări grafice, ele au nume speciale (exprimate de obicei prin prescurtări de 2 sau 3 litere). Dintre caracterele de control uzuale, pot fi amintite:

- CR (Carriage Return). Este interpretat de perifericele de ieșire, provocând mutarea cursorului pe ecran la începutul liniei curente. La imprimantă, provoacă mutarea carului de tipărire la începutul liniei;
- LF (Line Feed). Provoacă trecerea la linie nouă (atât la imprimantă, cât și pe ecran);
- TAB (Horizontal Tabulation). Provoacă deplasarea cursorului pe ecran sau a carului de tipărire cu un număr predefinit de poziții la dreapta sau într-o poziție predefinită;
- BS (Backspace). Provoacă deplasarea cursorului cu o poziție înapoi (la stânga) sau ștergerea caracterului de la stânga cursorului;
- DEL (Delete). Provoacă ștergerea caracterului indicat de cursor;
- BEL (Bel). Provoacă emiterea unui sunet de avertizare;
- FF (Form Feed). Provoacă avansul hârtiei (formularului) la pagină nouă (la imprimantă).

O serie de coduri de control se folosesc în transmisiile de date, având semnificații specifice. De exemplu, **ACK** (**Acknowledge**) se folosește pentru confirmarea transmisiei corecte a unui bloc de date, iar **NAK** (**Negative Acknowledgement**) pentru semnalarea unei transmisii incorecte.

Dintre caracterele tipăribile uzuale, se pot aminti:

- Literele mari 'A' ... 'Z'
- Literele mici 'a' ... 'z'
- Cifrele zecimale '0' ... '9'
- Spațiul (SP), care are codul 20H și este tipărit ca un spațiu liber (de fapt nu se tipărește nimic, ci se lasă o pauză); caracterul SP (spațiu, ' ') se numește și blank.

Valorile numerice ale codurilor ASCII se exprimă prin 2 cifre hexazecimale, în care prima cifră este limitată la 7F).

$2^4 2^5 2^6 \Rightarrow$ $2^3 2^2 2^1 2^0 \Downarrow$	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Tabelul 1.8 Codul ASCII standard

1.4 Tipuri de date utilizate în limbaj de asamblare

Limbajul de asamblare 80x86 operează cu anumite tipuri de date de bază. Aceste tipuri sunt recunoscute de către procesor și fac parte integrantă din formatul anumitor instrucțiuni mașină. De asemenea, limbajul dispune de directive specifice pentru definirea acestor tipuri de date.

Caracteristic pentru un tip de date este domeniul de valori, care rezultă atât din tipul de date în sine, cât și dintr-o eventuală interpretare funcție de context. Tipurile de date sunt:

BYTE (1 octet)

Acest tip de date ocupă 1 octet și poate fi reprezentat atât în memoria internă, cât și într-un registru de 8 biți al procesorului. Interpretările tipului byte pot fi:

- întreg pe 8 biți cu sau fără semn;
- caracter ASCII.

Directiva pentru definirea datelor de acest tip este DB (Define Byte).

WORD (2 octeți)

Acest tip de date ocupă 2 octeți și poate fi reprezentat atât în memoria internă, cât și într-un registru de 16 biți al procesorului. Interpretările tipului word pot fi:

- întreg pe 16 biți cu sau fără semn;
- secvență de două caractere ASCII;
- adresă de memorie de 16 biți.

Directiva pentru definirea datelor de acest tip este DW (Define Word). Partea mai puțin semnificativă este memorată la adrese mici. De exemplu, dacă presupunem întregul 1234H la adresa 1000H, atunci octetul 34H se va găsi la adresa 1000H, iar octetul 12H la adresa 1001H. Similar, se memorează și secvențele de două caractere ASCII.

DOUBLE-WORD (4 octeți)

Acest tip de date ocupă 4 octeți și poate fi reprezentat atât în memoria internă, cât și într-o pereche de registre de 16 biți sau într-un registru de 32 de biți (la procesoarele de 32 de biți). Interpretările tipului dword pot fi:

- întreg pe 32 de biți cu sau fără semn;
- număr real în simplă precizie;
- adresă de memorie de 32 de biți.

Directiva pentru definirea datelor de acest tip este DD (Define Double-Word). Valorile mai puțin semnificative se memorează la adrese mici. În cazul adreselor pe 32 de biți, adresa de segment este memorată la adrese mari, iar deplasamentul (offsetul), la adrese mici.

QUAD-WORD (8 octeți)

Acest tip de date ocupă 8 octeți și poate fi reprezentat atât în memoria internă, cât și într-o pereche de registre de 32 de biți (numai la procesoarele de 32 de biți). Interpretările tipului qword pot fi:

- întreg pe 64 de biți cu sau fără semn;
- număr real în dublă precizie;

Directiva pentru definirea datelor de acest tip este DQ (Define Quad-Word).

TEN-BYTES (10 octeți)

Acest tip de date ocupă 10 octeți și poate fi reprezentat atât în memoria internă, cât și într-unul din registrele coprocesoarelor matematice 80x87. Interpretările tipului tbyte pot fi:

- număr întreg reprezentat ca secvență de cifre BCD (împachetate), cu semn memorat explicit;
- număr real în precizie extinsă.

Directiva pentru definirea datelor de acest tip este DT (Define Ten-Bytes).

În reprezentarea întregilor, se consideră numere cu maxim 19 cifre zecimale. Un grup de două cifre se reprezintă pe 1 octet, partea mai puțin semnificativă fiind la adrese mici. În cadrul unui octet, cifra BCD mai semnificativă este în partea high a octetului. Cele 19 cifre BCD ocupă așadar 76 de biți. Ultimul grup de 4 biți (partea high a octetului aflat la adresa cea mai mare) este destinat memorării semnului. De fapt, semnul se memorează doar pe bitul cel mai semnificativ al acestui ultim octet. Asambloarele acceptă și numere cu 20 de cifre zecimale, atât timp cât cifra cea mai semnificativă, care se va reprezenta pe ultimul grup de 4 biți, nu intră în conflict cu bitul de semn.

Teoretic, valoarea maximă corect reprezentabilă este:

```
+ 9999 99999 99999 99999
```

iar valoarea minimă este:

```
- 9999 99999 99999 99999
```

dar se acceptă și valori de genul:

```
+ 79999 99999 99999 99999
- 79999 99999 99999 99999
```

în care cifra cea mai semnificativă este reprezentată doar pe 3 biți.

Din motive de siguranță, este bine să ne limităm la valorile garantate, adică la cel mult 19 cifre zecimale și semn.

Să considerăm următorul exemplu de program:

```
.model small
.data
      b1    db    -1, 10, 17H, 0FFH
      b3    db    'a', 'b'
      b2    db    "abcdef", 0

      w1    dw    1234H, -1, 'AB'
      w2    dw    w1

      d1    dd    12345678H, -1
      d2    dd    1.0, -1.0, 0.5
      d3    dd    d1

      q1    dq    1000000000000002H, -1
      q2    dq    1.0, -1.0

      t1    dt    12345678900000012345
      t2    dt    -12345678900000012345
      t3    dt    99999999999999999
      t4    dt    -99999999999999999
      t5    dt    79999999999999999
      t6    dt    -79999999999999999
      t7    dt    1.0
end
```

În acest exemplu, este definit doar un modul de date, folosind directivele și tipurile de date existente în limbaj. Liniile care încep cu un punct sunt directive care stabilesc modelul de memorie (.model) și, respectiv, definesc un segment de date (.data). Directiva end marchează sfârșitul programului.

Se definesc date cu cele 5 tipuri de directive (db, dw, dd, dq, dt), asociindu-se acestora și nume simbolice (b1, w2 etc.). Fișierul listing obținut în urma asamblării acestui program are (între altele) următorul conținut:

```
Turbo Assembler   Version 2.0 03/31/96 23:08:59 Page 1
      A.ASM
1  0000                          .model small
2  0000                          .data
3  0000 FF 0A 17 FF              b1    db    -1, 10, 17H, 0FFH
4  0004 61 62                    b3    db    'a', 'b'
5  0006 61 62 63 64 65 66 00     b2    db    "abcdef", 0
6
7  000D 1234 FFFF 4142           w1    dw    1234H, -1, 'AB'
8  0013 000Dr                    w2    dw    w1
9
10 0015 12345678 FFFFFFFF        d1    dd    12345678H, -1
11 001D 3F800000 BF800000+       d2    dd    1.0, -1.0, 0.5
12      3F000000
13 0029 00000015sr               d3    dd    d1
14
15 002D 1000000000000002+        q1    dq    1000000000000002H, -1
16      FFFFFFFFFFFFFFFF
17 003D 3FF0000000000000+        q2    dq    1.0, -1.0
18      BFF0000000000000
19
20 004D 01234567890000012345     t1    dt    1234567890000012345
21 0057 81234567890000012345     t2    dt    -1234567890000012345
22 0061 09999999999999999999     t3    dt    9999999999999999999
23 006B 89999999999999999999     t4    dt    -9999999999999999999
24 0075 79999999999999999999     t5    dt    7999999999999999999
25 007F F9999999999999999999     t6    dt    -7999999999999999999
26 0089 3FFF8000000000000000     t7    dt    1.0
27                          end
```

Prima coloană listează numărul liniei din fișierul sursă. Apoi este listată adresa și conținutul câmpului de date corespunzător unei linii sursă. Dacă acest conținut este listat pe mai multe linii, se folosește semnul + pentru a indica aceasta. Se remarcă reprezentările numerelor reale și ale întregilor negativi. Simbolurile r și s indică faptul că este vorba de o adresă relativă (deplasament) sau de segment. De exemplu, variabila d3 conține adresa de segment 0000 și deplasamentul 0015H, adică adresa relativă (pe 32 de biți) a variabilei d1. La întregii BCD pe 10 octeți, se observă memorarea explicită a bitului de semn: reprezentările pentru t1 și t2 diferă doar prin acest bit.

Listingul nu indică și poziția fizică a octeților în memorie. Următorul listing reprezintă zona de memorie corespunzătoare modulului de date de mai sus:

```
ds:0000 FF 0A 17 FF 61 62 61 62 63 64 65 66 00 34 12 FF
ds:0010 FF 42 41 0D 00 78 56 34 12 FF FF FF FF 00 00 80
ds:0020 3F 00 00 80 BF 00 00 00 3F 15 00 68 53 02 00 00
ds:0030 00 00 00 00 10 FF FF FF FF FF FF FF FF 00 00 00
ds:0040 00 00 00 F0 3F 00 00 00 00 00 00 F0 BF 45 23 01
ds:0050 00 00 89 67 45 23 01 45 23 01 00 00 89 67 45 23
ds:0060 81 99 99 99 99 99 99 99 99 99 09 99 99 99 99 99
ds:0070 99 99 99 99 89 99 99 99 99 99 99 99 99 99 79 99
ds:0080 99 99 99 99 99 99 99 99 F9
```

Din examinarea acestui listing, se observă, de exemplu, reprezentarea internă a variabilei double-word d1, aflată la adresa 0015H: secvența de octeți este 78 56 34 12, deci la adrese mici se găsește partea mai puțin semnificativă. Similar, variabila de tip word 'AB' (aflată la adresa 11H), este memorată prin secvența de octeți 42H 41H. Acest listing este obținut la încărcarea modulului de date în memorie, când adresele de segment sunt relocate, deci corespund unor adrese fizice concrete. Adresa variabilei d1 (conținută în d3) apare aici în forma (5368:0015). Pentru valorile reale, se regăsesc reprezentările descrise în 1.3.3.

Variabila t1 (aflată la adresa 004DH) este reprezentată prin secvența de octeți: 45 23 01 00 00 89 67 45 23 01, deci cifrele mai puțin semnificative sunt la adrese mici, iar în cadrul unui octet, cifra mai semnificativă este în partea high. Variabila t2 (aflată la adresa 0057H) este reprezentată prin secvența: 45 23 01 00 00 89 67 45 23 81, ceea ce arată că bitul de semn (reprezentat prin cifra 8) este în partea high a ultimului octet din reprezentare.

Valorile maxime și minime sunt reprezentate la adresele 0061H și 006BH, prin secvențele: 99 99 99 99 99 99 99 99 99 09, respectiv: 99 99 99 99 99 99 99 99 99 89.

Se observă și reprezentarea valorilor t5 și t6, în forma **99 99 99 99 99 99 99 99 99 79** și, respectiv, **99 99 99 99 99 99 99 99 99 F9**, în care apare cifra cea mai semnificativă 7, cu bitul de semn 0 sau 1.

Trebuie precizat că formatele de reprezentare pentru date numerice (întregi sau reali) se regăsesc în aceeași formă și în limbajele de nivel înalt.

În implementarea Borland PASCAL, există caractere și întregi pe 2, 4 și 8 octeți, care corespund tipurilor de date byte, word, double word și quad-word. De asemenea, există tipuri de date în virgulă mobilă pe 4, 8 și 10 octeți, în aceeași formă de reprezentare internă ca cea descrisă în 1.3.3.

În implementarea Borland C există tipul char (1 octet), tipuri întregi pe 2 și 4 octeți, cu sau fără semn (int, long, unsigned int, unsigned long) și tipuri în virgulă

mobilă pe 4, 8 și 10 octeți (float, double, long double), cu aceleași reprezentări interne ca cele din 1.3.3.

1.4 Arhitectura procesorului 8086. Formarea adresei fizice

Arhitectura procesorului 8086, din punctul de vedere al programului utilizator, este ilustrată schematic în Figura 1.2. Sunt figurate registrele accesibile prin program.

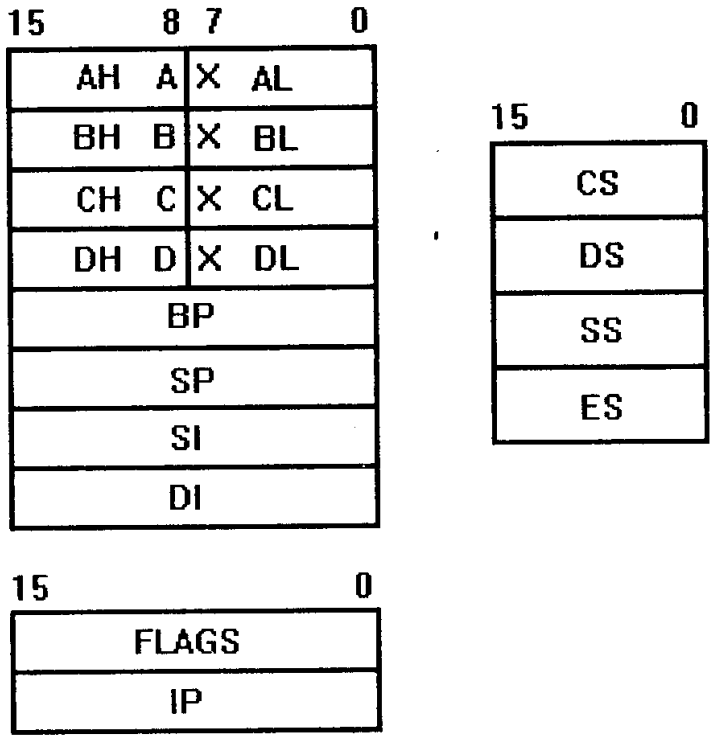


Figura 1.2 Registrele procesorului 8086

Toate registrele sunt de 16 biți. O serie de registre (AX, BX, CX, DX) sunt disponibile și la nivel de octet, părțile mai semnificative fiind AH, BH, CH și DH, iar cele mai puțin semnificative, AL, BL, CL și DL. Denumirile registrelor sunt:

- AX - registru acumulator
- BX - registru de bază general
- CX - registru contor
- DX - registru de date
- BP - registru de bază pentru stivă (base pointer)
- SP - registru indicator de stivă (stack pointer)
- SI - registru index sursă
- DI - registru index destinație

Registrul notat FLAGS cuprinde flagurile procesorului sau ale bistabililor de condiție, iar registrul IP (instruction pointer) este contorul program.

Denumirile registrelor de segment sunt:

- CS - registru de segment de cod (code segment)
- DS - registru de segment de date (data segment)
- SS - registru de segment de stivă (stack segment)
- ES - registru de segment de date suplimentar (extra segment)

Se observă că denumirile registrelor de segment corespund zonelor principale ale unui program executabil. Astfel, perechea de registre (CS:IP) va indica totdeauna adresa următoarei instrucțiuni care se va executa, iar perechea (SS:SP) indică totdeauna adresa vârfului stivei. Registrele DS și ES sunt folosite pentru a accesa date.

Procesorul 8086 dispune de adrese pe 20 de biți, fiind capabil să adreseze 1 megaoctet de memorie (2^{20}). Se pune problema cum se formează adresa fizică pe 20 de biți (deci pe 5 cifre hexa), deoarece toate registrele procesorului sunt de 16 biți, putând codifica adrese în domeniul 0000÷0FFFFH (pe 4 cifre hexa), deci într-un spațiu de maxim 64 KO.

Memoria unui sistem cu procesor 8086 este divizată în segmente. Un segment este o zonă continuă de memorie, de lungime maximă de 64 KO, care începe la o adresă fizică multiplu de 4. Acest fapt înseamnă că ultima cifră hexa a adresei de început a unui segment este totdeauna 0. Ca atare, această cifră se poate omite și adresa de segment se poate reprezenta tot pe 16 biți. Adresele de început ale segmentelor se vor găsi întotdeauna într-unul din cele 4 registre de segment.

Adresarea în interiorul unui segment se realizează printr-un deplasament (offset) relativ la începutul segmentului. Deoarece un segment nu poate depăși 64 KO, deplasamentul se poate memora tot pe 16 biți. Deplasamentul poate fi o constantă sau conținutul unui registru care permite adresarea memoriei.

În concluzie, pentru adresarea unui octet de memorie, se folosesc două cantități pe 16 biți: o adresă de segment (conținută obligatoriu într-un registru de segment) și un deplasament. Deoarece ambele cantități sunt pe 16 biți, se vorbește de adrese (sau pointeri) de 32 de biți, deși adresa fizică este doar pe 20 de biți.

Formarea adresei fizice (pe 20 de biți) este realizată automat (prin hardware) de către o componentă a procesorului, conform Figurii 1.3.

Concret, adresa fizică se obține prin deplasarea adresei de segment cu 4 biți la stânga și prin adunarea deplasamentului. Pentru specificarea unei adrese complete (de 32 de biți), se folosește notația (segment:offset) sau

(registru_segment:offset). De exemplu, putem specifica o adresă prin (18A3:5B27) sau prin (DS:5B27).

Trebuie remarcat faptul că asocierea (segment:offset) - adresă fizică nu este biunivocă, deoarece la o aceeași adresă fizică pot să corespundă mai multe perechi (segment:offset). De exemplu, perechile (18A3:5B27) și (18A2:5B37) reprezintă aceeași adresă fizică. În situația în care deplasamentul este redus la minim, adică în domeniul 0÷F, corespondența devine biunivocă. Adresele de acest tip se numesc adrese normalizate sau pointeri normalizați.

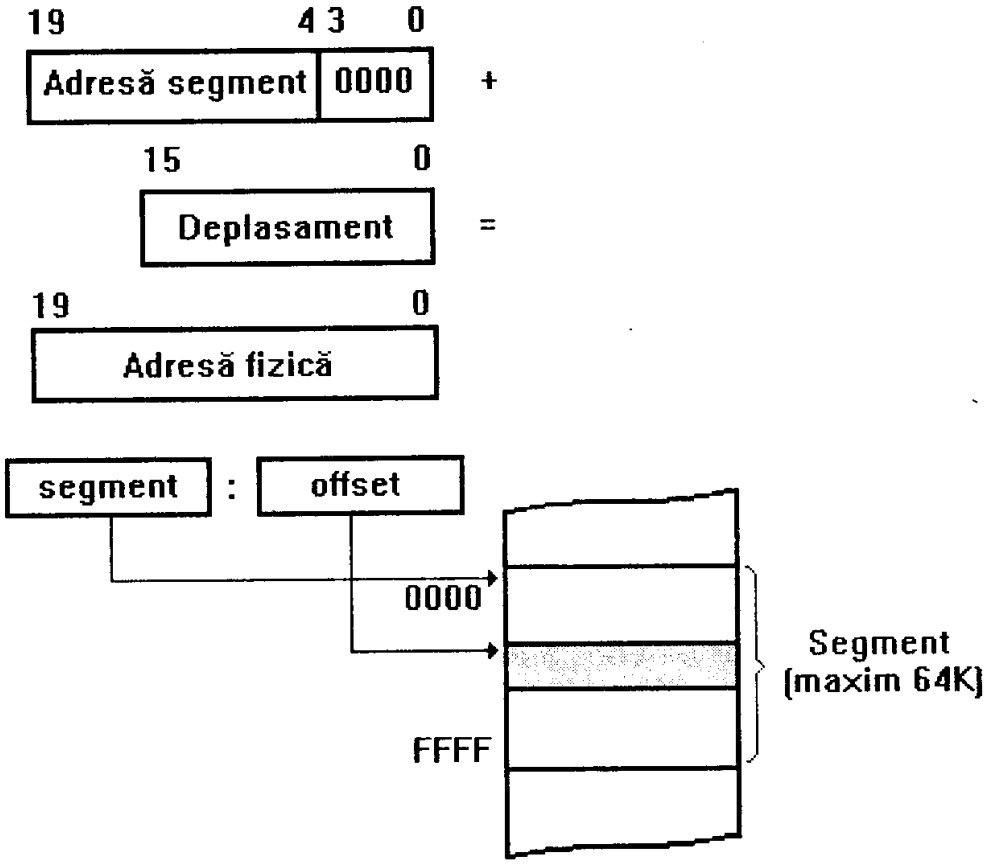


Figura 1.3 Formarea adresei fizice

O adresă completă de 32 de biți este memorată cu offsetul la adrese mici și cu adresa de segment la adrese mari. Adresele complete se pot obține cu directiva DD (Define Double-Word).

Registrul de flaguri (**bistabili de condiție**) al procesorului 8086 are configurația din Figura 1.4. O serie de flaguri sunt flaguri aritmetice: acestea sunt poziționate la 0 sau la 1 ca urmare a unor operații aritmetice sau logice. Celelalte flaguri controlează anumite operații ale procesorului.

Semnificația flagurilor este următoarea:

- **CF (Carry Flag, bistabil de transport)** - semnifică un transport sau un împrumut din/în bitul cel mai semnificativ al rezultatului, de exemplu la operații de adunare sau de scădere.
- **PF (Parity Flag, flag de paritate)** - este poziționat în așa fel încât numărul de biți egali cu 1 din octetul cel mai puțin semnificativ al rezultatului, împreună cu flagul PF, să fie impar; altfel formulat, suma modulo 2 a tuturor biților din octetul c.m.p.s. și a lui PF să fie 1.
- **AF (Auxiliarry Carry Flag, bistabil de transport auxiliar)** - indică un transport sau un împrumut din/în bitul 4 al rezultatului.
- **ZF (Zero Flag, bistabil de zero)** - este poziționat la 1 dacă rezultatul operației este 0.
- **SF (Sign Flag, bistabil de semn)** - este poziționat la 1 dacă b.c.m.s. al rezultatului (bitul de semn) este 1.
- **OF (Overflow Flag, bistabil de depășire)** - este poziționat la 1 dacă operația a condus la o depășire de domeniu a rezultatului (la operații cu sau fără semn).

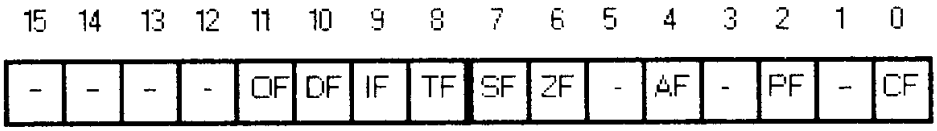


Figura 1.4 Registrul de flaguri al procesorului 8086

- **TF (Trap Flag, bistabil de urmărire)** - dacă este poziționat la 1, se forțează o întrerupere, pe un nivel predefinit, la execuția fiecărei instrucțiuni; acest fapt este util în programele de depanare, în care este posibilă rularea pas cu pas a unui program.
- **IF (Interrupt Flag, bistabil de întreruperi)** - dacă este poziționat la 1, procesorul ia în considerație întreruperile hardware externe; altfel, acestea sunt ignorate.
- **DF (Direction Flag, bistabil de direcție)** - precizează sensul (crescător sau descrescător) de variație al adreselor la operațiile cu șiruri de octeți sau de cuvinte.

Flagurile CF, PF, AF, ZF, SF și OF sunt numite flaguri aritmetice. Flagurile TF, IF și DF sunt numite flaguri de control.

1.5 Moduri de adresare

Pentru a scrie un program ASM într-o formă cât mai simplă, se pot folosi directivele simplificate de definire a segmentelor. Acestea sunt:

- .model small (precizează un model de memorie)

- .code (definire de segment de cod)
- .stack n (definire de segment de stivă)
- .data (definire de segment de date)
- end etichetă (sfârșit logic al programului)

În principiu, segmentul de cod va cuprinde programul executabil (instrucțiuni), iar segmentul de date va cuprinde date definite de utilizator. Segmentul de stivă nu este folosit explicit. Comentariile se scriu folosind simbolul ;. Ceea ce urmează după ; până la sfârșitul liniei curente este considerat comentariu.

Instrucțiunile procesorului 8086 pot avea unul sau doi operanzi, care pot fi registre sau operanzi aflați în memorie. Când există doi operanzi, unul este obligatoriu un registru (deci nu pot fi doi operanzi în memorie).

Modurile de adresare specifică modul în care se calculează adresa operandului aflat în memorie. Se folosesc denumirile de **adresă de segment** (AS) pentru adresa de început a segmentului în care se găsește operandul și **adresă efectivă** (AE), pentru deplasamentul operandului în cadrul segmentului respectiv. Adresa de segment și adresa efectivă formează **adresa fizică** (AF), conform mecanismului descris în 1.4. Adresa de segment este furnizată de unul din cele 4 registre de segment.

Operanzii pot fi de tip octet sau de tip word, iar operațiile se pot face numai între operanzi de același tip. Tipul operandului aflat în memorie rezultă, de obicei, din context (din registrele folosite). În situațiile în care acest lucru nu este posibil, se folosește operatorul ptr al limbajului de asamblare, în forma byte ptr sau word ptr, pentru a explicita tipul operandului.

- **Adresare imediată**

În acest mod de adresare, operandul apare chiar în instrucțiune. De exemplu:

```
mov    ax, 1          ; Pune in AX valoarea 1
add    bx, 2          ; Aduna 2 la BX
```

- **Adresare directă**

Adresa efectivă a operandului este furnizată printr-un deplasament în interiorul segmentului curent (vezi Figura 1.5).

```
.data
      VAL    dw 1
.code
      mov    bx, VAL      ; Pune in BX continutul locatiei
                          ; VAL in formatul instructiunii,
                          ; expresia VAL se va inlocui cu
                          ; deplasamentul in cadrul
                          ; segmentului de date
      add    cx, [100]    ; Deplasament explicit
```

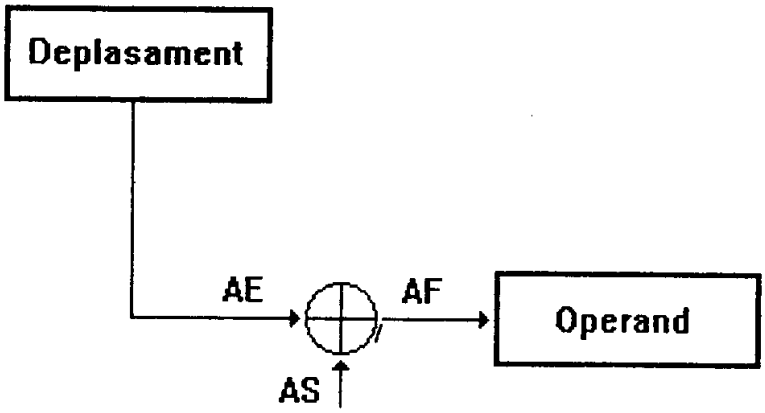


Figura 1.5 Adresare directă

Prima instrucțiune înseamnă „pune conținutul locație VAL în BX", ceea ce va conduce la o valoare a registrului BX egală cu 1. A doua instrucțiune înseamnă „adună la CX conținutul locației de memorie de la adresa efectivă 100". De observat prezența parantezelor drepte: fără ele, instrucțiunea ar fi însemnat „adună valoarea 100 la registrul CX". În ambele instrucțiuni, operandul aflat în memorie este considerat de tip word.

Acest mod de adresare folosește registrul DS ca registru implicit de segment.

- **Adresare indirectă (prin registre)**

Adresa efectivă a operandului este dată de conținutul registrelor BX, SI sau DI (vezi Figura 1.6). Registrul implicit de segment este DS.

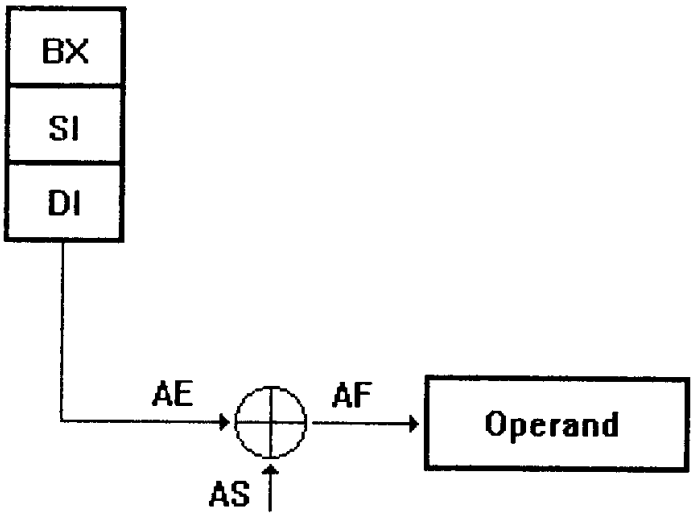


Figura 1.6 Adresare indirectă

În textul sursă se scrie, de exemplu:

```
mov    ax, [bx]                ; Pune in AX continutul
                               ; locatiei de memorie de la
                               ; adresa data de BX
mov    [di], cx                ; Pune continutul lui CX in
                               ; locatia de la adresa data
                               ; de DI.
add    byte ptr [si], 2        ; Aduna 2 la octetul aflat la
                               ; adresa data de SI
```

În primele două instrucțiuni, operandul aflat în memorie este considerat de 16 biți (datorită celuilalt operand, care este un registru de 16 biți). Dacă a treia instrucțiune ar fi scrisă:

```
add    [si], 2
```

asamblorul nu ar ști tipul operandului din memorie. Instrucțiunea poate fi interpretată ca „adună 2 la octetul de la adresa dată de SI" sau „adună 2 la cuvântul de la adresa dată de SI" și s-ar obține un mesaj de eroare la asamblare. Folosirea operatorului **ptr** elimină această ambiguitate.

- **Adresare bazată sau indexată**

Adresa efectivă se obține adunând la unul din registrele de bază (BX sau BP) sau la unul din registrele index (SI sau DI) un deplasament constant pe 8 sau 16 biți (vezi Figura 1.7). Registrul de segment implicit este DS (dacă se folosesc BX, SI sau DI) și, respectiv, SS (dacă se folosește BP).

Forma de memorare a deplasamentului (pe 8 sau 16 biți) este determinată de asamblor. În textul sursă, se scrie pur și simplu un deplasament constant.

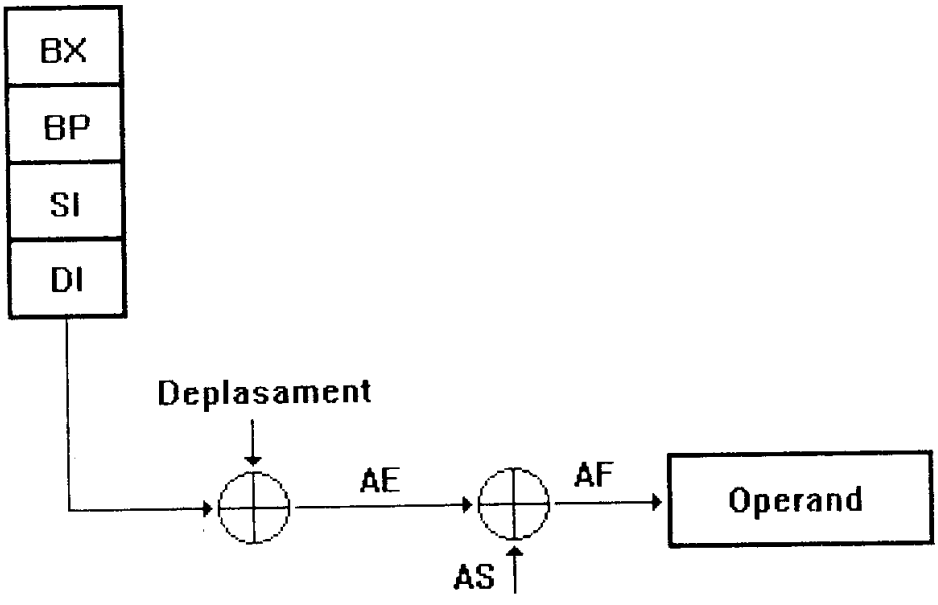


Figura 1.7 Adresare bazată sau indexată

Se pot folosi diverse forme de scriere, ca în exemplul de mai jos, în care bx este (din punct de vedere al mecanismului de adresare) adresă de bază, iar VAL este un deplasament constant. Din punct de vedere al programului, care își propune să acceseze elementele tabloului de octeți VAL, semnificația este exact inversă: VAL este adresa de început a tabloului (deci o adresă de bază), iar bx este un indice (un deplasament).

```
.data
      VAL   dw    10 dup (0)   ; Definire de tablou de 10
                                ; octeti
.code
      mov   bx, 5
      mov   ax, VAL[bx]
      mov   ax, bx[VAL]
      mov   ax, [bx+VAL]
      mov   ax, [bx].VAL
```

Este posibil și alt mod de acces, în care se încarcă registrul BX cu deplasamentul tabloului VAL (folosind operatorul offset) și se folosește un indice explicit.

```
      mov   bx, offset VAL
      mov   ax, 5[bx]
      mov   ax, bx[5]
      mov   ax, [bx+5]
      mov   ax, [bx].5
```

În această situație, adresa de bază (conținută în BX) este adresa de început a tabloului VAL.

Trebuie remarcat că, în textul sursă, toate formele de adresare se scriu cu operatorul de indexare (paranteze drepte). De exemplu, se poate scrie:

```
mov   ax, [bx]                  ; Adresare indirecta
mov   ax, [100]                 ; Adresare directa
mov   ax, [bx+100]              ; Adresare bazata
```

De remarcat că folosirea registrului BP cu un deplasament nul permite o formă de scriere asemănătoare cu adresarea indirectă, dar codificată intern ca adresare bazată:

```
mov   ax, [bx]                  ; Adresare indirecta
mov   ax, [bp]                  ; Adresare bazata cu
                                ; deplasament nul
```

- **Adresare bazată și indexată**

Adresa efectivă este formată prin adunarea unuia din registrele de bază (BX sau BP) cu unul din registrele index (SI su DI) și cu un deplasament de 8 sau 16 biți (vezi Figura 1.8).

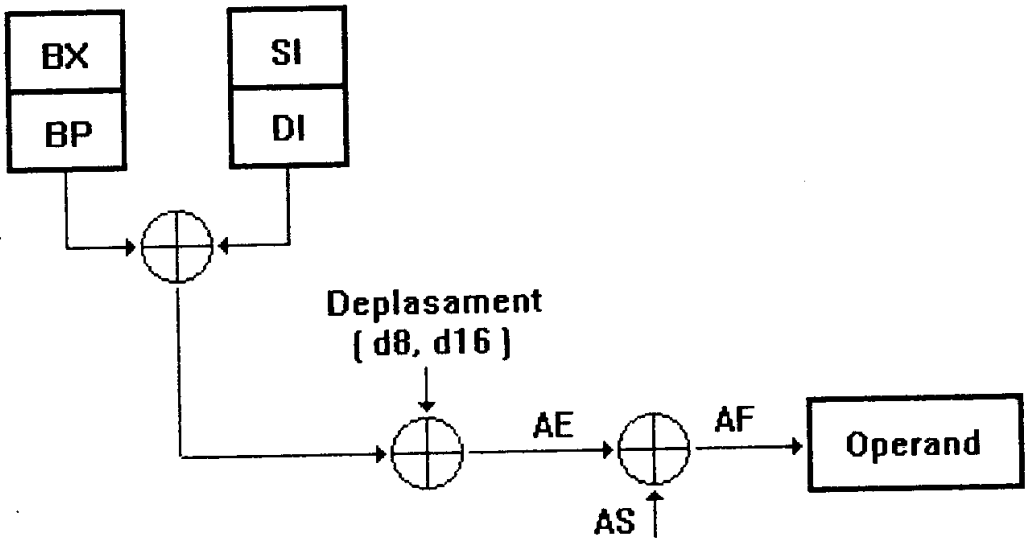


Figura 1.8 Adresare bazată și indexată

Registrele de segment implicite sunt DS (dacă se folosește BX cu SI sau cu DI) și, respectiv, SS (dacă se folosește BP cu SI sau cu DI). Deplasamentul poate fi și nul. În textul sursă se folosește operatorul de indexare [ ]:

```
mov    ax, [bx][si]
mov    ax, [bx+si+7]
mov    ax, [bx+si].7
mov    ax, [bp][di][7]
```

În descrierea modurilor de adresare, s-a precizat de fiecare dată registrul implicit de segment care participă la formarea adresei fizice a operandului aflat în memorie. Acest registru implicit este DS în toate modurile de adresare care nu implică registrul BP și, respectiv, SS, în modurile de adresare în care participă registrul BP.

Aceste reguli implicite pot fi modificate prin folosirea prefixelor de segment. Un prefix de segment este un octet care apare înaintea codului instrucțiunii mașină și care identifică explicit registrul de segment folosit. În textul sursă, se scrie un registru de segment, urmat de :, înaintea operandului aflat în memorie:

```
mov    bx, ds:[bp+7]
mov    ax, cs:[si][bx+3]
mov    ax, ss:[bx]
```

Evident, gestiunea a ceea ce se găsește în segmentele adresate în acest fel cade în sarcina programatorului.

1.6 Codificarea internă a instrucțiunilor

Instrucțiunile procesorului 8086 pot fi codificate pe un număr de octeți cuprins între 1 și 6. Codificarea cuprinde codul instrucțiunii (tipul operației), modul de

adresare, deplasamentul operandului aflat în memorie (pe 8 sau 16 biți) sau adresa efectivă (în cazul adresării directe), date (operanzi) imediate, pe 8 sau 16 biți. Pe lângă formatul propriu-zis, mai poate exista un prefix de segment sau un prefix de repetare.

Formatul general al instrucțiunilor 8086 este ilustrat în Figura 1.9.

La instrucțiunile cu doi operanzi, un operand este obligatoriu de tip registru și este codificat de câmpul REG, iar celălalt este registru sau operand în memorie și este codificat de câmpul R/M sau de câmpurile R/M și MOD. Semnificația câmpurilor din Figura 1.9 este următoarea:

- **Câmpul D (destinație)** - codifică sensul operației:

  D = 0 R/M ← R/M operație REG

  D = 1 REG ← REG operație R/M

De exemplu, codificările instrucțiunilor:

```
add   [bx], si
add   si, [bx]
```

vor diferi numai prin câmpul D (aceeași operație și aceiași operanzi, dar diferă sensul).

- **Câmpul W (word)** - codifică lungimea operanzilor (octet sau cuvânt):

  W = 0 operație la nivel de octet

  W = 1 operație la nivel de cuvânt

De exemplu, instrucțiunile:

```
add   ax, bx
add   al, bl
```

vor diferi doar prin câmpul W.

- **Câmpul REG** identifică operandul de tip registru.
- **Câmpul R/M** identifică al doilea operand de tip registru sau un registru care participă la formarea adresei efective a operandului aflat în memorie.
- **Câmpul MOD** codifică modul de adresare:

  MOD = 11 înseamnă că al doilea operand este tot de tip registru(codificat de câmpul R/M).

  MOD = 00 înseamnă adresare directă (R/M = 6), indirectă (R/M = 4, 5, 6) sau adresare bazată și indexată cu deplasament nul (R/M = 0, 1, 2, 3). În aceste moduri de adresare, octeții 3 și 4 din Figura 1.9 lipsesc din formatul instrucțiunii (cu excepția cazului R/M = 6, adică al adresării directe).

  MOD = 01 înseamnă adresare bazată (R/M = 6, 7), indexată (R/M = 4, 5) sau bazată și indexată (R/M = 0, 1, 2, 3), toate cu un deplasament de 8

biți. În aceste moduri de adresare, octetul 4 din Figura 1.9 lipsește din formatul instrucțiunii.

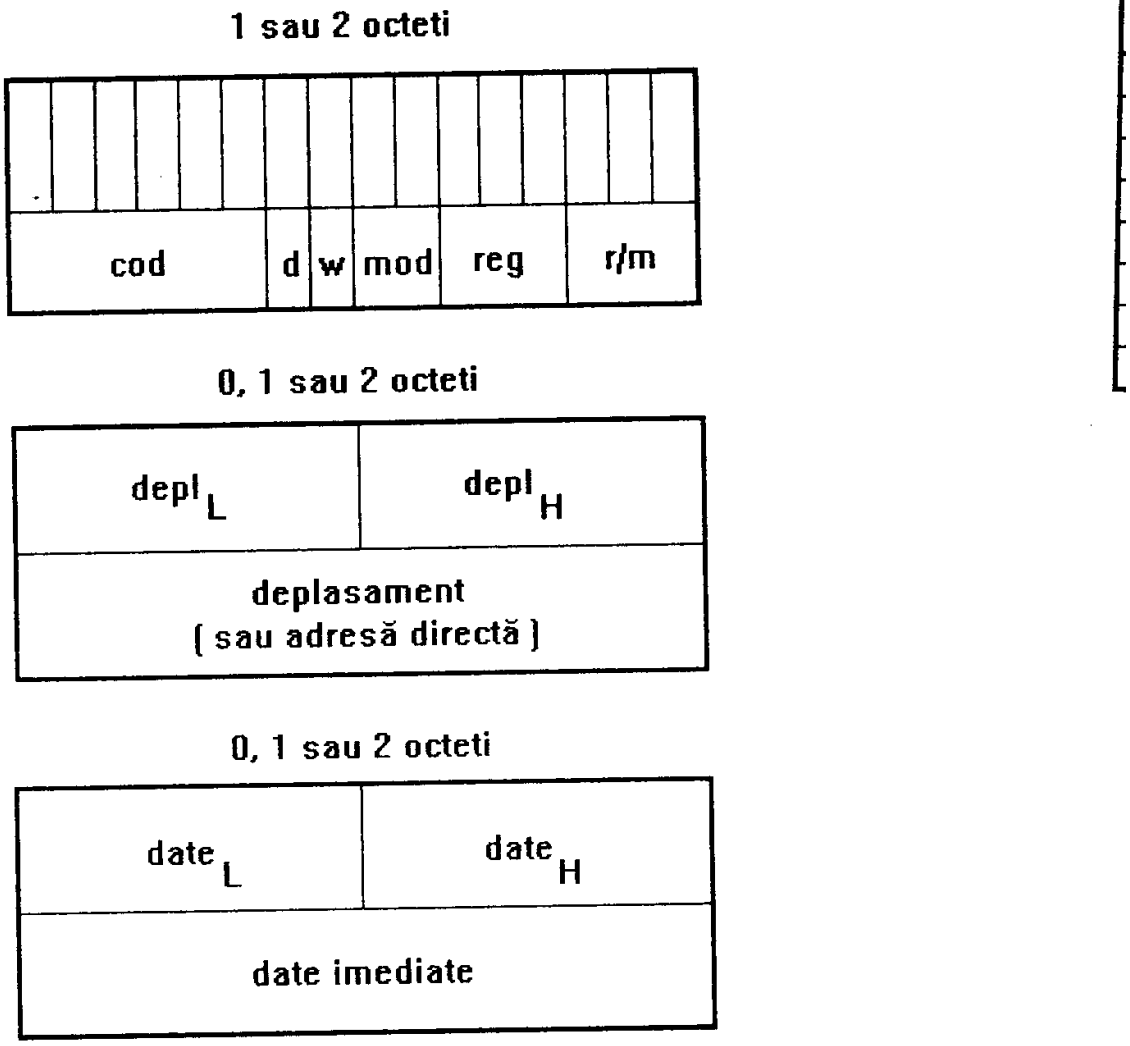


Figura 1.9 Formatul instrucțiunilor 8086

MOD = 10 este asemănător cu MOD = 01, dar deplasamentele sunt pe 16 biți. În aceste moduri de adresare, octeții 3 și 4 din Figura 1.9 sunt prezenți în formatul instrucțiunii.

Situațiile de mai sus sunt descrise sintetic în Tabelul 1.10.

MOD →	11		00	01	10
R/M ↓	W = 0	W = 1			
0	AL	AX	(BX)+(SI)	(BX)+(SI)+d8	(BX)+(SI)+d16
1	CL	CX	(BX)+(DI)	(BX)+(DI) +d8	(BX)+(DI) +d16
2	DL	DX	(BP)+(SI)	(BP)+(SI) +d8	(BP)+(SI) +d16
3	BL	BX	(BP)+(DI)	(BP)+(DI) +d8	(BP)+(DI) +d16
4	AH	SP	(SI)	(SI) +d8	(SI) +d16
5	CH	BP	(DI)	(DI) +d8	(DI) +d16
6	DH	SI	directă	(BP) +d8	(BP) +d16
7	BH	DI	(BX)	(BX) +d8	(BX) +d16

Tabelul 1.10 Codificarea modurilor de adresare

Capitolul 2

Setul de instrucțiuni 8086

Procesoarele din familia Intel dispun de un set puternic de instrucțiuni aritmetice, logice, de transfer și de control, ceea ce permite încadrarea acestor procesoare în clasa **CISC** (**Complex Instruction Set Computer**).

În cadrul acestui capitol, sunt prezentate în detaliu instrucțiunile de bază ale familiei de procesoare Intel. În general, pentru toate tipurile de instrucțiuni, se poate folosi orice mod de adresare (inclusiv cu prefixe de segment) expuse în capitolul anterior. Acolo unde este cazul, se specifică tipurile interzise de adresare.

Setul de instrucțiuni este grupat în 6 clase:

- **instrucțiuni de transfer**, care deplasează date între memorie sau porturi de intrare/ieșire și registrele procesorului, fără a executa nici un fel de prelucrare a datelor;
- **instrucțiuni aritmetice și logice**, care prelucrează date în format numeric;
- **instrucțiuni pentru șiruri**, specifice operațiilor cu date alfanumerice;
- **instrucțiuni pentru controlul programului**, care în esență se reduc la salturi și la apeluri de proceduri;
- **instrucțiuni specifice întreruperilor** hard și soft;
- **instrucțiuni pentru controlul procesorului**.

Această împărțire este realizată după criterii funcționale. De exemplu, instrucțiunile PUSH și POP sunt considerate ca instrucțiuni de transfer, deși, la prima vedere, ar putea fi considerate instrucțiuni specifice procedurilor. Același lucru despre instrucțiunile IN și OUT, care interfațează procesorul cu lumea exterioară: ele sunt considerate instrucțiuni de transfer, deși ar putea fi considerate instrucțiuni de intrare/ieșire. Intrările și ieșirile sunt însă cazuri particulare de transfer.

Fiecare categorie de instrucțiuni este însoțită de specificarea explicită a bistabililor de condiție care sunt modificați în urma execuției.

2.1 Instrucțiuni de transfer

Instrucțiunile de transfer presupun o copiere a unui octet sau a unui cuvânt de la o sursă la o destinație. Această copiere (atribuire) va fi desemnată uneori printr-o săgeată, de la sursă către destinație. Destinația poate fi un registru, o locație de memorie sau un port de ieșire, iar sursa poate fi un registru, o locație

de memorie, date imediate (constante) sau un port de intrare. Cu excepția cazului important al instrucțiunilor PUSH și POP (2.1.1), sursa și destinația nu pot fi simultan locații de memorie.

Instrucțiunile de transfer se clasifică în instrucțiuni generale, specifice acumulatorului, specifice adreselor și specifice flagurilor.

În specificarea destinației și a sursei, se vor folosi notațiile segment:offset pentru adrese și notația (x), pentru a desemna „conținutul lui x". Notația cu paranteze se poate extinde pe mai multe nivele. De exemplu, (BX) înseamnă conținutul registrului BX, iar ((BX)) înseamnă conținutul locației de memorie adresate de BX (implicit prin DS). Similar ES:((BX)) va însemna conținutul locației de memorie adresate de registrele ES și BX.

Cu excepția instrucțiunilor SAHF și POPF (2.1.4), nici o instrucțiune de transfer nu modifică vreun bistabil de condiție.

2.1.1 Instrucțiuni de transfer generale - MOV, PUSH, POP, XCHG

Instrucțiunea MOV (Move Data - Transferă date)

Formatul general este:

```
MOV   destinatie, sursa          ; (destinatie) ← sursa
```

în care sursă și destinație (operanzii) sunt octeți sau cuvinte respectând regulile descrise mai sus.

Următoarele operații sunt ilegale:

- sursa și destinația nu pot fi ambele operanzi în memorie;
- nu pot fi folosite registrele FLAGS și IP;
- operanzii nu pot avea dimensiuni diferite;
- registrul CS nu poate apărea ca destinație.

Procesorul original 8086 are restricțiile suplimentare:

- nu pot fi transferate date imediate într-un registru de segment;
- operanzii nu pot fi simultan registre de segment.

Aceste ultime două restricții au fost relaxate la variantele ulterioare (80286 și peste). Exemple de instrucțiuni corecte:

```
MOV   AX, BX
MOV   AL, CH
MOV   VAL[BX][SI], AL
MOV   byte ptr [BX+100], 5
```

Ultima instrucțiune de mai sus ar fi fost ambiguă fără utilizarea operatorului ptr: MOV [BX+100], 5 se poate interpreta ca „pune valoarea 5 în octetul de la adresa

DS:BX+100" sau, la fel de bine, „pune valoarea 5 la cuvântul de la adresa DS:BX+100". Forma „byte ptr" precizează că este vorba de un transfer pe octet.

Exemple de instrucțiuni incorecte:

```
MOV   AL, BX              ; operanzi de lungime diferita
MOV   [BX], [SI]          ; ambii operanzi in memorie
MOV   CS, AX              ; registrul cs apare ca destinatie
```

De reținut că o instrucțiune de forma:

```
.data
      ALFA  DB    1
.code
      MOV   AL, ALFA
```

încarcă în AL conținutul locației de memorie ALFA. Dacă se dorește încărcarea adresei efective a variabilei ALFA, se poate folosi operatorul OFFSET:

```
MOV   BX, OFFSET ALFA
```

sau instrucțiunea LEA (2.1.3).

Instrucțiunea PUSH (Push Data - Salvează date în stivă)

Forma generală este:

```
PUSH  sursa
```

în care sursă este un operand pe 16 biți (registru general de 16 biți, registru de segment sau locație de memorie), iar semnificația este „copiază sursă în vârful stivei". Concret, execuția instrucțiunii se face după secvența:

```
(SP)  ←     (SP) - 2
SS : ((SP)+1 : (SP))    ←     sursa
```

ceea ce înseamnă că se decrementează SP cu 2 și în octeții de la adresele (SP)+1 și (SP) din segmentul de stivă se copiază operandul sursă. Copierea respectă regula de memorare a cantităților pe mai mulți octeți și anume, partea mai puțin semnificativă (partea low) se memorează la adrese mici. O exprimare detaliată a instrucțiunii ar putea fi:

```
(SP)  ←     (SP) - 2
SS : ((SP)+1)     ←     high (sursa)
SS : ((SP))       ←     low (sursa)
```

Exemple de instrucțiuni corecte:

```
PUSH  BX
PUSH  ES
PUSH  [BX]
PUSH  [BP+5]
```

```
PUSH  ES:[BX][SI+4]
```

Exemplu de instrucțiuni incorecte:

```
PUSH  AL            ; Operand pe 1 octet
```

Instrucțiunea POP (Pop Data - Refă date din stivă)

Forma generală este:

```
POP   destinatie
```

în care destinație este un operand pe 16 biți (registru general de 16 biți, registru de segment sau locație de memorie), iar semnificația este „copiază conținutul vârfului stivei în destinație". Registrul CS nu poate apărea ca destinație. Concret, execuția instrucțiunii se face după secvența:

```
destinatie  ←    SS : ((SP)+1 : (SP))
(SP)        ←    (SP) + 2
```

ceea ce înseamnă că se transferă octeții de la adresele (SP)+1 și (SP) din segmentul de stivă în operand (destinație) și apoi se incrementează SP cu 2. O exprimare detaliată a instrucțiunii ar putea fi:

```
high (destinatie) ←    SS : ((SP)+1)
low (destinatie)  ←    SS : ((SP))
(SP)              ←    (SP) + 2
```

Exemple de instrucțiuni corecte:

```
POP   BX
POP   ES
POP   ES:[DI]
POP   [BP+5]
POP   SS:[BX+4]
```

Exemple de instrucțiuni incorecte:

```
POP   AL            ; Operand pe 1 octet
POP   CS            ; Registrul cs
```

Din analiza instrucțiunilor PUSH și POP, reiese că o secvență de refaceri ale unor cantități salvate în stivă (de exemplu, conținutul unor registre) trebuie scrisă în ordine inversă. Dacă secvența de salvare a fost:

```
PUSH  AX
PUSH  BX
PUSH  CX
```

atunci secvența de refacere trebuie să fie:

```
POP   CX
```

```
POP    BX
POP    AX
```

Dacă registrele de mai sus conțin valorile AX = 1234H, BX = 5678H și CX = 9ABCH, iar registrul SP conține (înainte de salvări) valoarea 1288H, atunci imaginea stivei va fi cea din Figura 2.1.

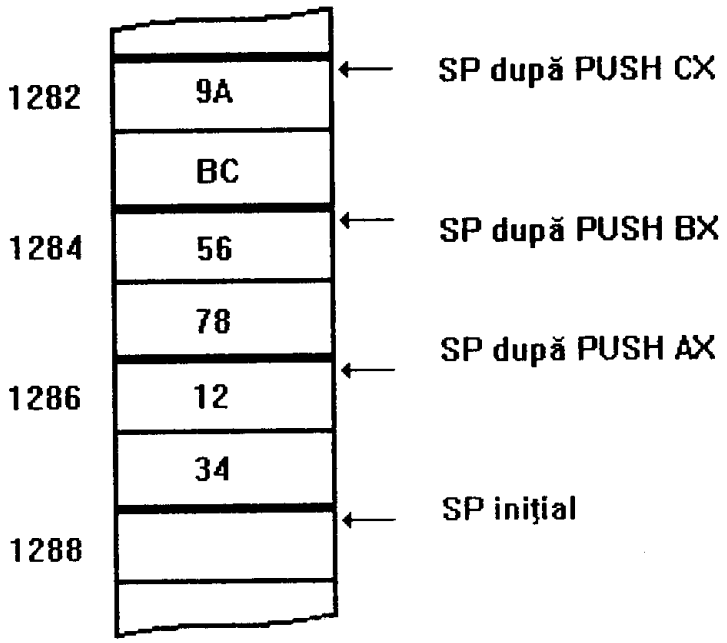


Figura 2.1 Imaginea stivei după o secvență de instrucțiuni PUSH și POP

La operațiile cu stiva, trebuie avut grijă ca o secvență de refacere să aducă indicatorul SP la valoarea de dinainte de secvența de salvare. Acest lucru înseamnă că, de regulă, numărul operațiilor POP trebuie să coincidă cu cel al operațiilor PUSH.

Tot ca regulă generală, nu este indicat să se modifice explicit locațiile aflate „în josul stivei", adică la valori mai mari sau egale cu valoarea curentă a registrului SP. În acele locații se pot afla informații a căror alterare ar putea compromite definitiv execuția programului. Trebuie, deci, folosite cu atenție secvențele de genul:

```
MOV    BX, SP
MOV    SS:[BX], AX
MOV    byte ptr SS:[BX+2], 1
```

Instrucțiunile PUSH și POP se mai pot folosi la transferul indirect al unor registre. Secvența:

```
PUSH  DS
POP   ES
```

copiază conținutul registrului DS în ES, lăsând indicatorul SP neschimbat.

Instrucțiunile PUSH și POP realizează de fapt un transfer din memorie în memorie. Dacă operandul din cele două instrucțiuni este o locație de memorie, ceea ce transferă la PUSH este conținutul acelei zone. Secvența:

```
.data
      X      dw 100
.code
      PUSH   X
```

va pune în vârful stivei valoarea 100 (conținutul lui X). Dacă se dorește punerea în stivă a adresei efective sau complete a variabilei X, se va folosi instrucțiunea MOV și operatorul OFFSET sau instrucțiunea LEA (vezi 2.1.3), respectiv registrul de segment care adresează curent segmentul respectiv. Dacă segmentul în care e definită variabila nu este adresat în mod curent de nici un registru de segment, se poate folosi operatorul SEG, care furnizează segmentul în care este definit operandul:

```
.code
      LEA    AX, X
      PUSH   AX                    ; Offset-ul la adrese mici
      PUSH   DS                    ; Apoi segmentul
;
;     Sau ...
;
      MOV    AX, OFFEST X
      PUSH   AX                    ; Offset-ul lui X
      MOV    AX, SEG X
      PUSH   AX                    ; Adresa de segment a lui X
```

Instrucțiunea XCHG (Exchange Data - Interschimbă date)

Forma generală este:

```
XCHG  destinatie, sursa
```

iar semnificația este cea de interschimbare a sursei cu destinația. Registrele de segment nu pot apărea ca operanzi și, bineînțeles, cel puțin un operand trebuie să fie registru.

Exemple de instrucțiuni corecte:

```
XCHG  AL, AH
XCHG  BX, SI
XCHG  ES:[BX], AX
```

Exemple de instrucțiuni incorecte:

```
XCHG  AL, BX              ; Operanzi de lungime diferita
```

```
XCHG  ES, AX                ; Registru de segment
```

Instrucțiunea XCHG este utilă la interschimbarea a două cantități aflate în memorie. Dacă op1 și op2 sunt doi operanzi aflați în memorie, care trebuie interschimbați, secvența standard de interschimbare (folosind un registru general reg) este:

```
MOV   reg, op1
XCHG  reg, op2
MOV   op1, reg
```

2.1.2 Instrucțiuni de transfer specifice acumulatorului - IN. OUT, XLAT

Instrucțiunea IN (Input Data - Citește date de la port de intrare)

Forma generală este:

```
IN    destinatie, port
```

în care destinație este registrul AL sau AX, iar port este fie o constantă cuprinsă între 0 și 255, fie registrul DX. Variantele noi de procesoare acceptă orice registru în locul lui AL sau AX. Semnificația este că se execută o citire de la portul de intrare specificat, pe 8 sau 16 biți, după cum s-a specificat registrul AL sau AX. La variantele noi de procesoare (80286 și peste), registrele AL sau AX pot fi substituite cu orice registre generale.

Instrucțiunea OUT (Output Data - Scrie date la port de ieșire)

Forma generală este:

```
OUT   destinatie, port
```

în care destinație este registrul AL sau AX, iar portul de ieșire este specificat la fel ca la instrucțiunea IN.

Instrucțiunile IN și OUT sunt singurele instrucțiuni propriu-zise care pot realiza interacțiunea procesorului cu alte dispozitive. Unele arhitecturi de calculatoare au organizată memoria în așa fel încât zone din spațiul adresabil sunt dedicate unor echipamente periferice, și nu unor zone propriu-zise de memorie. Accesul la zonele respective de memorie va însemna de fapt un acces la echipamentul periferic. Asemenea sisteme de intrări/ieșiri se numesc de tip „memory-mapped" (intrări-ieșiri organizate ca spațiu de memorie).

Să considerăm că un echipament periferic necesită un port de stare și un port de date, ambele pe 8 biți. Într-un sistem de intrări-ieșiri obișnuit, vor exista două porturi de intrare, de exemplu, 0F8H și 0F9H, dedicate perifericului respectiv. Într-un sistem de tip „memory-mapped", vor exista două adrese, de obicei adiacente, de exemplu c800:0000 și c800:0001, corespunzătoare porturilor de

stare și de date. Secvențele de citire stare, respectiv date, în cele două tipuri de intrări/ieșiri vor fi:

```
IN    AL, 0F8H     ; Citire stare
IN    AL, 0F9H     ; Citire date
MOV   ES, 0C800H
MOV   AL, ES:[0]   ; Citire stare
MOV   AL, ES:[1]   ; Citire date
```

Instrucțiunea XLAT (Translate - Translatează)

Instrucțiunea nu are operanzi, iar semnificația este:

```
(AL)  ←     DS : ((BX) + (AL))
```

adică se aduce în AL conținutul octetului de la adresa (BX)+(AL). Această instrucțiune este folosită împreună cu tabele de translatare (de unde și numele), utile în conversia unor tipuri de date. De exemplu, dacă dorim să convertim o valoare numerică între 0 și 15 la cifra hexa corespunzătoare, putem folosi secvența:

```
.data
      TABELA       DB     '0123456789ABCDEF'
.code
      MOV          BX, OFFSET TABELA
      XLAT
```

prin care se încarcă în BX offsetul tabelei de octeți și se face apoi translatarea.

2.1.3 Instrucțiuni de transfer specifice adreselor - LEA, LDS, LES

Aceste instrucțiuni transferă o adresă efectivă într-un registru general sau o adresă completă pe 32 de biți într-o pereche de registre.

Instrucțiunea LEA (Load Effective Address - Încarcă adresa efectivă)

Forma generală este:

```
LEA   registru, sursa
```

în care sursa este un operand aflat în memorie, specificat printr-un mod oarecare de adresare. Efectul este copierea adresei efective a operandului (offset-ul în cadrul segmentului) în registrul general specificat. Exemple:

```
LEA   BX, ALFA
LEA   DI, ALFA [BX] [SI]
```

Un calcul similar al adresei efective se obține și cu operatorul OFFSET și instrucțiunea MOV, calcul care se face însă la asamblare. Astfel, instrucțiunea LEA permite și moduri de adresare bazată și/sau indexată.

Instrucțiunea LEA se folosește pentru încărcarea registrelor de bază sau de segment cu adresele efective ale unor operanzi din memorie, în vederea unor adresări ulterioare. Secvența:

```
.data
      VECTOR      DW 10, 20, 30, 40, 50
.code
      LEA   BX, VECTOR
      MOV   SI, 4
      MOV   AX, [BX][SI]
```

va încărca în AX al treilea element al tabloului VECTOR.

Instrucțiunile LDS (Load Data Segment - Încarcă DS) și LES (Load Extra Segment - Încarcă ES)

Forma generală este:

```
LDS   reg, sursa
LES   reg, sursa
```

în care reg este un registru general de 16 biți, iar sursa este un operand de tip double-word aflat în memorie, care conține o adresă completă de 32 de biți. Efectul instrucțiunii este:

```
(reg)        ←     ((sursa))
(DS)/(ES)    ←     ((sursa)+2)
```

adică se încarcă perechea DS:reg, respectiv ES:reg, cu o adresă completă de 32 de biți. De obicei, instrucțiunea LDS se folosește împreună cu directiva Define DoubleWord, ca în exemplul următor:

```
.data
      x      db     10
      y      db     15
      adr_x  dd     x
      adr_y  dd     y
.code
      LDS    SI, adr_x
      LES    DI, adr_y
      MOV    byte ptr [SI], 20
      MOV    byte ptr ES:[DI], 30
```

Variabilele adr_x și adr_y, care conțin adresele far ale variabilelor x și y, sunt încărcate în DS:SI și ES:DI. Se accesează apoi variabilele x și y, folosind adresarea indexată.

Următorul exemplu utilizează o tabelă de adrese. Să presupunem că avem 8 variabile word diferite, numite V_0, ..., V_7 și dorim să încărcăm în AX variabila a cărui indice este dat de registrul CX. Cu alte cuvinte, dacă CX=0, încărcăm

V_0, dacă CX=1, încărcăm V_1 etc. Definim în acest scop tabela de adrese TAB_ADR și folosim adresarea indexată.

```
.data
      TAB_ADR      dd     V_0, V_1, V_2, V_3, V_4, V_5, V_6, V_7
.code
      MOV    BX, CX
      ADD    BX, BX
      ADD    BX, BX
      LES    DI, TAB_ADR[BX]
      MOV    AX, ES:[DI]
```

Deoarece o poziție din tabelă ocupă 4 octeți, valoarea indicelui din CX trebuie înmulțită cu 4. Acest lucru se obține prin două instrucțiuni ADD BX, BX, care adună BX la el însuși.

2.1.4 Instrucțiuni de transfer specifice flagurilor (LAHF, SAHF, PUSHF, POPF)

Instrucțiunea LAHF (Load AH with FLAGS - Încarcă AH cu FLAGS)

Instrucțiunea nu are operanzi, iar semnificația este dată de denumire: se încarcă registrul AH cu partea mai puțin semnificativă a registrului de flaguri:

$$AH \leftarrow FLAGS_{0:7}$$

Instrucțiunea SAHF (Store AH into FLAGS - Depune AH în FLAGS)

Este perechea instrucțiunii LAHF, semnificația fiind:

$$FLAGS_{0:7} \leftarrow AH$$

Instrucțiunea PUSHF (Push Flags - Salvează FLAGS în stivă)

Nu are operanzi, iar efectul este plasarea registrului FLAGS în vârful stivei („salvarea" flagurilor în stivă):

```
(SP)  ←     (SP) - 2
SS:((SP) + 2 : (SP))     ←     FLAGS
```

Instrucțiunea POPF (Pop Flags - Reface FLAGS din stivă)

Este perechea instrucțiunii PUSHF:

```
FLAGS ←     SS:((SP) + 1 : (SP))
(SP)  ←     (SP) + 2
```

Instrucțiunile LAHF și SAHF citesc și scriu explicit partea mai puțin semnificativă a registrului de flaguri. Partea mai semnificativă poate fi citită și modificată prin mici artificii cu instrucțiunile PUSHF și POPF. Secvența:

```
PUSHF
POP   AX
```

va încărca în AX registrul de flaguri, deci în AH vom avea partea mai semnificativă, iar secvența:

```
PUSH  AX
POPF
```

va încărca AX în registrul de flaguri.

Cu excepția instrucțiunilor explicite SAHF și POPF, nici o instrucțiune de transfer nu modifică bistabilii de condiție.

2.2 Instrucțiuni aritmetice și logice

A doua categorie importantă de instrucțiuni o constituie instrucțiunile aritmetice și logice. Rezultatul operației este totdeauna depus într-unul din operanzi. La instrucțiunile cu doi operanzi, rezultatul este depus în primul operand. Instrucțiunile modifică flagurile CF, AF, ZF, SF, PF și OF, motiv pentru care acestea mai sunt numite și flaguri aritmetice.

2.2.1 Semnificația flagurilor aritmetice

- **Flagul Carry** = 1 semnifică un transport/împrumut din/în bitul cel mai semnificativ (b.c.m.s.) al rezultatului, unde b.c.m.s. poate fi bitul 7 sau bitul 15. Interpretarea acestui transport sau împrumut este cea de depășire la operațiile de adunare/scădere cu operanzi fără semn.
- **Flagul Auxiliarry Carry** = 1 semnifică un transport/împrumut din/în bitul 4 al rezultatului, fiind utilizat la operațiile cu numere BCD.
- **Flagul Zero** este 1 dacă rezultatul operației este nul.
- **Flagul Sign** este 1 dacă bitul de semn al rezultatului (bitul 7 sau 15) este 1.
- **Flagul Parity** este 1 dacă suma modulo 2 a celor 8 biți mai puțin semnificativi ai rezultatului este 0.
- **Flagul Overflow** este 1 dacă operația a condus la un transport înspre b.c.m.s., dar nu din b.c.m.s. al rezultatutului, sau invers. Altfel, OF se poziționează la zero. Similar, în cazul împrumutului: OF = 1 dacă a avut loc un împrumut dinspre b.c.m.s., dar nu din exterior, sau invers. Situațiile descrise sunt ilustrate în Figura 2.2.

Interpretarea flagului OF este aceea de depășire la operațiile de adunare/scădere cu operanzi cu semn.

Să considerăm valoarea pe 1 octet 0FFH și să adunăm 1 la această valoare. Rezultatul este 0 și transport atât din bitul 7, cât și din bitul 6 în bitul 7. Astfel, CF = 1 și OF = 0. Considerând operanzii fără semn (255 și 1), adunarea lor

provoacă depășire, ceea ce corespunde cu CF = 1. Dacă se consideră operanzii cu semn, (-1 și 1), adunarea lor nu provoacă depășire și OF = 0.

Să considerăm, în mod similar, valorile 70H și 10H și suma acestora. Rezultatul este 80H și nu apare transport din bitul 7, dar este transport din bitul 6 în bitul 7. Ca atare, CF = 0 și OF = 1. În interpretarea fără semn, valorile operanzilor sunt 112 și 16, suma lor fiind 128, deci nu apare depășire (CF = 0). În interpretarea cu semn, valorile sunt tot 112 și 16, dar suma lor depășește domeniul admisibil al octeților cu semn și OF = 1.

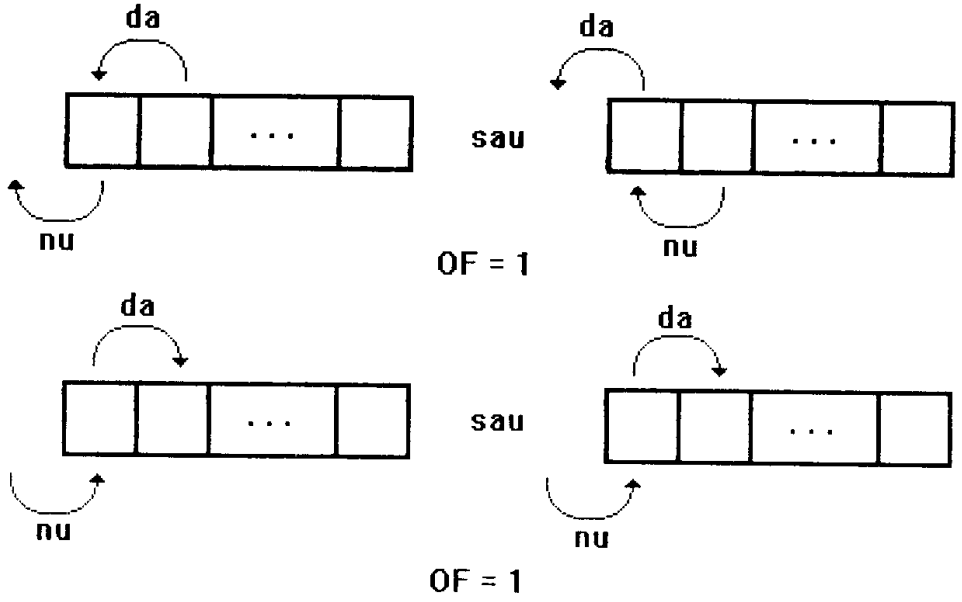


Figura 2.2 Poziționarea bistabilului Overflow

În fine, să considerăm operanzii 0FDH și 02H și suma lor. La adunare, se obține valoarea 0FFH și nu apare transport nici din bitul 7, nici din bitul 6. Ca atare, CF = 0 și OF = 0. Valorile fără semn sunt 253 și 2, iar suma lor (255) este în domeniul admisibil, deci CF = 0. Valorile cu semn sunt -3 și 2, iar suma lor (-1) este în domeniul de reprezentare al numerelor cu semn, deci OF = 0.

La fiecare instrucțiune aritmetică sau logică, se vor preciza explicit flagurile afectate, adică flagurile care se poziționează conform rezultatului. Un flag neafectat rămâne la vechea valoare. Există și situații în care unele flaguri au valori nedefinite.

2.2.2 Instrucțiuni specifice adunării (ADD, ADC, INC, DAA, AAA)

Instrucțiunea ADD (Add - Adună)

Are forma generală:

```
ADD   destinatie, sursa
```

în care destinație poate fi un registru general sau o locație de memorie, iar sursă poate fi un registru general, o locație de memorie sau o valoare imediată. Cei doi operanzi nu pot fi simultan locații de memorie. Semnificația este:

```
(destinatie)  ←  (destinatie) + (sursa)
```

Flaguri afectate: AF, CF, PF, SF, ZF, OF (toate).

Operanzii pot fi pe 8 sau 16 biți și trebuie să aibă aceeași dimensiune. În caz de ambiguitate în ceea ce privește lungimea operanzilor, se folosește operatorul **ptr**. Iată câteva exemple:

```
ADD   AX, BX
ADD   AX, 1                    ; Lungime 16 biti
ADD   AL, 1                    ; Lungime 8 biti
ADD   word ptr [DI], -1        ; Dest. in memorie, sursa
                               ; imediata
ADD   ALFA, 3                  ; ALFA este definit cu DW
ADD   byte ptr ALFA, 3         ; Fortare instructiune
                               ; pe octet
```

Instrucțiunea ADC (Add with Carry - Adună cu transport)

Forma generală este:

```
ADC   destinatie, sursa
```

Formatul este similar cu instrucțiunea ADD, ceea ce diferă fiind adunarea bistabilului CF:

```
(destinatie)  ←  (destinatie) + (sursa) + (CF)
```

Flaguri afectate: AF, CF, PF, SF, ZF, OF (toate).

Adunarea cu Carry se folosește la adunări de operanzi pe mai multe cuvinte, în care poate apărea un transport intermediar. De exemplu, secvența de adunare a două numere pe 4 octeți este:

```
.data
      ALFA  dd    145A789FH
      BETA  dd    92457ABCH
      REZ   dd    ?
.code
      MOV   AX, word ptr ALFA       ; Cuvintele m.p.s. la
                                    ; adr. mici
      ADD   AX, word ptr BETA       ; Aici poate aparea
                                    ; transport
      MOV   word ptr REZ, AX
      MOV   AX, word ptr ALFA+2     ; Cuvintele m.s. la
                                    ; adr. mari
      ADC   AX, word ptr BETA+2     ; Se ia in considerare
```

```
                                        ; transportul precedent
        MOV   word ptr REZ+2, AX
```

Instrucțiunea INC (Increment - Incrementează)

Are forma generală:

```
INC   destinatie
```

în care destinație este un registru sau un operand în memorie, de tip octet sau cuvânt. Semnificația este:

```
(destinatie) ← (destinatie) + 1
```

Flaguri afectate: AF, PF, SF, ZF, OF (fără CF).

Instrucțiunea DAA (Decimal Adjust for Addition - Corecție zecimală după adunare)

Instrucțiunea nu are operanzi și efectuează corecția zecimală a acumulatorului AL, după o adunare cu operanzi în format BCD despachetat. Semnificația este:

```
daca (AL0:3) > 9 sau (AF) = 1, atunci {
      (AL) ← (AL) + 6
      (AF) ← 1
}
daca acum (AL4:7) > 9 sau CF = 1, atunci {
      (AL) ← (AL) + 60H
      (CF) ← 1
}
```

Flaguri afectate: CF, AF, CF, PF, SF, ZF. Flagul OF este nedefinit.

Dacă cifra BCD mai puțin semnificativă (AL0:3) este mai mare decât 9, deci este incorectă, sau dacă a avut loc un transport din bitul 3 în bitul 4, se corectează această cifră prin adunarea valorii 6. Dacă cifra BCD mai semnificativă este acum mai mare decât 9, se corectează similar această cifră. În ambele situații, se poziționează corespunzător AF și CF, pentru a indica depășirea care a avut loc.

Să considerăm, de exemplu, adunarea valorilor BCD 65 și 17. Aceste valori se reprezintă prin octeții 65H și 17H. În urma adunării, se obține rezultatul 7CH, care este incorect ca rezultat BCD. Operația de corecție (DAA) conduce la rezultatul 82H, ceea ce reprezintă suma BCD a celor două valori. Secvența de adunare trebuie să fie:

```
.data
      BCD1   db 65H
      BCD2   db 17H
      REZ    db ?
.code
```

```
MOV   AL, BCD1
ADD   AL, BCD2
DAA
MOV   REZ, AL
```

Instrucțiunea AAA (ASCII Adjust for Addition - Corecție ASCII a acumulatorului)

Instrucțiunea nu are operanzi și efectuează corecția acumulatorului AX, după operații de adunare cu operanzi BCD despachetați (o cifră BCD pe un octet). Semnificația este următoarea:

```
dacă (AL0:3) > 9 sau (AF) = 1, atunci {
      (AL) ← (AL) + 6
      (AH) ← (AH) + 1
      (AF) ← 1
      (CF) ← 1
      (AL) ← (AL) AND 0FH
}
```

Flaguri afectate: AF, CF, restul nedefinite.

Corecția se face tot prin adăugarea valorii 6 la (AL), dar se face și o incrementare a registrului (AH), în ideea că acolo s-ar putea ține cifra BCD despachetată mai semnificativă. Totodată se șterg biții 4:7 ai registrului AL, pentru a avea o cifră BCD despachetată. Să considerăm că registrele AX și BX conțin valorile 0309H și 0104H, ceea ce ar corespunde valorilor BCD despachetate 39 și 14. În urma adunării, se obține rezultatul 040DH, care este incorect. Instrucțiunea AAA, corectează acest rezultat la 0503H, care este suma corectă a celor două valori inițiale:

```
MOV   AX, 0309h
MOV   BX, 0104H
ADD   AX, BX          ; AX = 040DH
AAA                   ; AX = 0503H
```

2.2.3 Instrucțiuni specifice scăderii (SUB, SBB, DEC, NEG, CMP, DAS, AAS)

Instrucțiunea SUB (Subtract - Scade)

Are forma generală:

```
SUB   destinatie, sursa
```

unde destinatie și sursa sunt la fel ca la instrucțiunea ADD. Semnificația este:

```
(destinatie) ← (destinatie) - (sursa)
```

Flaguri afectate: AF, CF, PF, SF, ZF, OF (toate).

Scăderea poate fi privită ca o adunare cu complementul față de 2 al operandului sursă, dar cu inversarea rolului bistabilului CF, în sensul că, dacă la această adunare echivalentă apare transport, atunci CF = 0 și reciproc. Să considerăm secvența de instrucțiuni:

```
MOV    AL, 1
SUB    AL, 05EH
```

Rezultatul este 0A3H și există împrumut, deci CF = 1. Dacă luăm complementul față de 2 al sursei, obținem valoarea 0A2H, care, adunată la destinație (adică la 1), conduce la valoarea 0A3H. La această adunare echivalentă nu există transport, deci, conform regulii de mai sus, operația de scădere de la care am pornit va conduce la CF = 1.

Instrucțiunea SBB (Subtract with Borrow - Scade cu împrumut)

Are forma generală:

```
SBB    destinatie, sursa
```

în care destinatie și sursa sunt la fel ca la instrucțiunea ADD. Semnificația este:

```
(destinatie) <-- (destinatie) - (sursa) - (CF)
```

deci se ia în considerare un eventual împrumut anterior.

Flaguri afectate: AF, CF, PF, SF, ZF, OF (toate).

Instrucțiunea SBB se utilizează la scăderi de operanzi pe mai multe cuvinte, ca în exemplul următor:

```
.data
      ALFA   dd 145A789FH
      BETA   dd 92457ABCH
      REZ    dd ?
.code
      MOV    AX, word ptr ALFA        ; Cuvintele m.p.s. la
                                      ; adr. mici
      SUB    AX, word ptr BETA        ; Aici poate aparea
                                      ; imprumut
      MOV    word ptr REZ, AX
      MOV    AX, word ptr ALFA+2      ; Cuvintele m.s. la
                                      ; adr. mari
      SBB    AX, word ptr BETA+2      ; Se ia in considerare
                                      ; imprumutul precedent
      MOV    word ptr REZ+2, AX
```

Instrucțiunea DEC (Decrement - Decrementează)

Are forma generală:

```
DEC    destinatie
```

în care destinație este la fel ca la instrucțiunea INC. Semnificația este:

```
(destinatie) ← (destinatie) - 1
```

Flaguri afectate: AF, PF, SF, ZF, OF (fără CF).

Instrucțiunea NEG (Negate - Schimbă semnul)

Are forma generală:

```
NEG    destinatie
```

în care destinație este un registru sau o locație de memorie, pe 8 sau pe 16 biți. Semnificația este:

```
(destinatie) ← 0 - (destinatie)
```

deci se face o schimbare a semnului operandului.

Flaguri afectate: AF, CF, PF, SF, ZF, OF (toate).

De observat că schimbarea semnului poate conduce uneori la aceeași valoare, în cazul depășirii domeniului admisibil. De exemplu, secvența:

```
MOV    AL, -128
NEG    AL
```

va lăsa registrul AL neschimbat (80H), deoarece 128 și -128 au aceeași reprezentare internă.

Instrucțiunea CMP (Compare - Compară)

Are forma generală:

```
CMP    destinatie, sursa
```

iar semnificația este execuția unei scăderi temporare (destinație) - (sursă), fără a se modifica vreun operand, dar cu poziționarea bistabililor de condiție.

Flaguri afectate: AF, CF, PF, SF, ZF, OF (toate).

Testând bistabilii de condiție, putem deduce relația dintre cei doi operanzi. De exemplu, instrucțiunea CMP AX, BX va provoca o scădere temporară (AX) - (BX). Dacă ZF = 1 înseamnă că (AX) = (BX). Dacă CF = 1 înseamnă că la scădere a apărut un împrumut, deci (AX) < (BX), dacă sunt considerate ca numere fără semn.

Instrucțiunea CMP se folosește, de obicei, împreună cu instrucțiuni de salt condiționat (vezi 2.4.3).

Instrucțiunea DAS (Decimal Adjust for Subtraction - Corecție zecimală după scădere)

Instrucțiunea nu are operanzi și execută corecția zecimală a acumulatorului AL, după operații de scădere cu numere în format BCD împachetat. Semnificația este:

```
daca (AL0:3) > 9 sau (AF) = 1, atunci {
      (AL) ← (AL) - 6
      (AF) ← 1
}
daca acum (AL4:7) > 9 sau CF = 1, atunci {
      (AL) ← (AL) - 60H
      (CF) ← 1
}
```

Flaguri afectate: AF, CF, PF, SF, ZF. Flagul OF este nedefinit.

Explicația operațiilor de mai sus este similară cu cea de la instrucțiunea DAA. De exemplu, în urma secvenței:

```
.code
      MOV   AL, 52H
      SUB   AL, 24H              ; AL = 2EH
      DAS                        ; AL = 28H
```

se obține în AL rezultatul corect 28H (28 = 52 - 24).

Instrucțiunea AAS (ASCII Adjust for Subtraction - Corecție ASCII după scădere)

Instrucțiunea nu are operanzi și efectuează corecția acumulatorului AX, după operații de scădere cu operanzi BCD despachetați (o cifră BCD pe un octet). Semnificația este următoarea:

```
daca (AL0:3) > 9 sau (AF) = 1, atunci {
      (AL) ← (AL) - 6
      (AH) ← (AH) - 1
      (AF) ← 1
      (CF) ← 1
      (AL) ← (AL) AND 0FH
}
```

Flaguri afectate: AF, CF, restul nedefinite.

Se observă analogia cu instrucțiunea AAA, specifică adunării în format BCD despachetat.

2.2.4 *Instrucțiuni specifice înmulțirii (CBW, CWD, MUL, IMUL, AAM)*

Operațiile de înmulțire se fac între acumulator și un al doilea operand. Rezultatul operației este pe 16 sau, respectiv, 32 de biți. Se folosesc următoarele noțiuni, definite diferit la operații de 8 sau 16 biți:

- acumulator - registrul AL, respectiv, AX
- acumulator extins - registrul AX, respectiv perechea de registre DX:AX
- extensia acumulatorului - registrul AH, respectiv DX
- extensia de semn a acumulatorului - conținutul registrului AL, respectiv AX, reprezentat, ca număr cu semn, pe o lungime dublă (16, respectiv 32 de biți).

Instrucțiunea CBW (Convert Byte to Word - Convertește octet la cuvânt)

Este fără operanzi și are următoarea semnificație:

```
daca (AL7) = 0, atunci
       (AH) ← 0
altfel
       (AH) ← 1
```

Flaguri afectate: nici unul.

Practic, se extinde bitul de semn din AL la întreg registrul AH. Acest lucru este echivalent cu reprezentarea lui AL în complement față de 2, pe un număr dublu de biți. De exemplu, dacă AL = -3 (0FDH), atunci instrucțiunea CBW va forța în AX valoarea 0FFFDH, care este chiar reprezentarea în complement față de 2 a valorii - 3.

Instrucțiunea CWD (Convert Word to DoubleWord - Convertește cuvânt la dublu-cuvânt)

Este fără operanzi și are următoarea semnificație:

```
daca (AX15) = 0, atunci
       (DX) ← 0
altfel
       (DX) ← 1
```

Flaguri afectate: nici unul.

Se extinde bitul de semn din AX la întreg registrul DX, obținându-se astfel o reprezentare a lui AX pe 32 de biți.

Prin instrucțiunile CBW și CWD, se obțin extensiile de semn ale acumulatorului în acumulatorul extins.

Instrucțiunea MUL (Multiply - Înmulțește fără semn)

Are forma generală:

```
MUL    sursa
```

în care sursă poate fi un registru sau o locație de memorie de 8 sau 16 biți. Variantele noi de procesoare acceptă și date imediate ca operand sursă. Rezultatul se obține pe un număr dublu de biți (16 sau 32). Semnificația este:

```
(acumulator extins) ← (acumulator) * (sursa)
```

în care ambii operanzi se consideră numere fără semn.

Mai precis, dacă sursă este pe octet:

```
(AH:AL) ← (AL) * (sursa)
```

iar dacă sursă este pe cuvânt:

```
(DX:AX) ← (AX) * (sursa)
```

Flaguri afectate: dacă extensia acumulatorului (adică AH sau DX) este diferită de 0, atunci CF și OF sunt 1, altfel CF și OF sunt 0. Restul flagurilor sunt nedefinite. Exemple:

```
.data
      ALFA  db 10H
      BETA  dw 200H
.code
      MOV   AL, 10H
      MUL   ALFA                ; (AX) ← (AL) * ALFA
      MOV   AX, 20H
      MUL   BETA                ; (DX:AX) ← (AX) * BETA
      MOV   AX, 100H
      MOV   BX, 20H
      MUL   BX                  ; (DX:AX) ← (AX) * (BX)
```

În situația în care un operand este de tip byte, iar celălalt de tip word, se convertește operandul de tip byte la word, ca număr fără semn, deci cu partea mai semnificativă 0. De exemplu, pentru a înmulți valorile ALFA și BETA, se poate scrie:

```
MOV   AL, ALFA
MOV   AH, 0               ; (AX) = 0010H
MUL   BETA                ; (DX:AX) = 2000H
```

Se observă că instrucțiunea MUL nu poate conduce la depășiri. Înmulțind fără semn cele mai mari valori posibile pe 8/16 biți, se obțin valori corecte pe 16/32 de biți. De exemplu, $(2^{16} - 1)\bullet(2^{16}-1) = 2^{32} - 2^{17} + 1$, care este o valoare reprezentabilă pe 32 de biți.

Instrucțiunea IMUL (Integer Multiply - Înmulțește cu semn)

Are forma generală:

```
IMUL   sursa
```

fiind similară cu instrucțiunea MUL. Deosebirea este că operația de înmulțire se face considerând operanzii numere cu semn.

Flaguri afectate: dacă extensia acumulatorului reprezintă extensia de semn a acumulatorului, atunci CF și OF se poziționează la 0. Altfel, CF și OF devin 1. Restul flagurilor sunt nedefinite.

Nu pot apărea situații de depășire. Înmulțind cele mai mari valori în modul, se obțin valori corect reprezentabile. De exemplu, 127•127 = 16129, care se reprezintă corect ca număr cu semn pe 16 biți. Similar, (-128)•(-128) = 16384.

Dacă sunt necesare conversii de la byte la word, se va folosi instrucțiunea CBW, ca în secvența următoare:

```
.data
      ALFA  db -113
      BETA  dw -147
      REZ   dd ?
.code
      MOV   AL, ALFA
      CBW                              ; Conversie la word
      MUL   BETA                       ; Inmultire pe 16 biti
      MOV   word ptr REZ, AX           ; Partea m.p.s. la
                                       ; adrese mici
      MOV   word ptr REZ+2, DX         ; Partea m.s. la
                                       ; adrese mari
```

Instrucțiunea AAM (ASCII Adjust for Multiply - Corecție ASCII după înmulțire)

Instrucțiunea nu are operanzi și efectuează o corecție a acumulatorului AX, după o înmulțire pe 8 biți cu numere în format BCD despachetat. Semnificația este:

```
(AH) ← (AL)/10
(AL) ← (AL) MOD 10
```

Flaguri afectate: PF, SF, ZF, restul nedefinite.

De exemplu, să considerăm secvența:

```
MOV   AL, 5
MOV   CL, 9
MUL   CL
AAM
```

În urma înmulțirii, registrul AX va conține valoarea 2DH (45). Corecția prin AAM conduce la AH = 4 și AL = 5, deci AX = 0405H, adică reprezentarea BCD despachetat pentru valoarea zecimală 45.

2.2.5 Instrucțiuni specifice împărțirii (DIV, IDIV, AAD)

Împărțirea presupune că deîmpărțitul este pe o lungime dublă decât împărțitorul. Prin împărțire se obțin câtul și restul, de lungime egală cu a împărțitorului.

Instrucțiunea DIV (Divide - Împarte fără semn)

Are forma generală:

```
DIV   sursa
```

în care sursă e un operand (registru sau locație de memorie) pe octet sau pe cuvânt. Procesoarele moderne acceptă și date imediate ca operand sursă. Semnificația este următoarea:

```
(acumulator) ← (acumulator extins) / (sursa)
(extensia acumulatorului) ← (acumulator extins) MOD
(sursa)
```

Detaliind operațiile, se obține, în cazul în care sursa este pe octet:

```
(AL) ← (AX) / (sursa)
(AH) ← (AX) MOD (sursa)
```

iar dacă sursa este pe cuvânt:

```
(AX) ← (DX:AX) / (sursa)
(DX) ← (DX:AX) MOD (sursa)
```

Flaguri afectate: toate flagurile sunt nedefinite.

Împărțirea poate conduce la depășiri. În situația în care câtul rezultă mai mare decât valoarea maximă reprezentabilă pe 8, respectiv 16 biți, sau dacă împărțitorul este 0, rezultatele sunt nedefinite și se generează o întrerupere soft pe nivelul 0 (Divide Overflow - Depășire la împărțire). De exemplu, în secvența:

```
MOV   AX, 1000
MOV   BL, 2
DIV   BL
```

câtul ar trebui să fie 500, valoare care nu se poate reprezenta pe un octet. Ca atare, apare depășire și se va genera o întrerupere pe nivelul 0. Rutina afectată acestui nivel de întrerupere oprește de obicei programul executabil și afișează un mesaj de eroare la consolă.

Să considerăm câteva exemple de operații de împărțire fără semn, care ilustrează pregătirea operanzilor în situațiile care nu corespund celor două tipuri de bază (împărțire cuvânt la octet și împărțire dublu-cuvânt la cuvânt):

```
.data
      B1     db ?
      B2     db ?
      W1     dw ?
      W2     dw ?
      D1     dd ?
.code
; impartire octet la octet
      MOV    AL, B1
      MOV    AH, 0
      DIV    B2                  ; AL = cat, AH = rest
;
; impartire cuvant la octet
      MOV    AX, W1
      DIV    B1                  ; AL = cat, AH = rest
;
; impartire dublu-cuvant la cuvant
      MOV    AX, word ptr D1
      MOV    DX, word ptr D1 + 2
      DIV    W1                  ; AX = cat, DX = rest
;
; impartire cuvant la cuvant
      MOV    AX, W1
      MOV    DX, 0
      DIV    W2                  ; AX = cat, DX = rest
;
; impartire dublu-cuvant la byte
      MOV    AX, word ptr D1
      MOV    DX, word ptr D1 + 2
      MOV    BL, B1
      MOV    BH, 0
      DIV    BX                  ; AX = cat, DX = rest
```

Instrucțiunea IDIV (Integer Divide - Împarte cu semn)

Are forma generală:

```
IDIV  sursa
```

în care sursa este la fel ca la instrucțiunea DIV. Semnificația este aceeași ca la instrucțiunea DIV, cu diferența că operanzii sunt considerați cu semn, iar împărțirea se face cu semn.

Flaguri afectate: toate flagurile sunt nedefinite.

În situația în care câtul rezultă în afara domeniului reprezentabil pe 8, respectiv 16 biți, sau dacă împărțitorul este 0, rezultatele sunt nedefinite și se generează o întrerupere soft pe nivelul 0. Concret, câtul trebuie să fie în domeniul

[-128, +127] la împărțire pe octet, respectiv în domeniul [-32768, +32767] la împărțire pe cuvânt. De exemplu, în secvența:

```
MOV   AX, 400
MOV   BL, 2
IDIV  BL
```

câtul ar trebuie să rezulte 200, valoare care nu se poate reprezenta ca număr cu semn pe un octet. Ca atare, va apărea o întrerupere pe nivelul 0.

În ceea ce privește calculul restului la împărțirea cu semn, acesta este totdeauna calculat astfel încât să fie îndeplinită identitatea următoare:

```
a = b*(a/b) + a MOD b
```

în care a/b și a MOD b sunt câtul și restul calculate de instrucțiunea IDIV. De exemplu, în secvența:

```
MOV   AX, -10
MOV   BL, -3
IDIV  BL              ; AL = 3, AH = 0FFH
```

câtul rezultă +3, iar identitatea de mai sus conduce la restul - 1. Așadar (-10) MOD (-3) = -1 și în registrul AH se va găsi 0FFH (-1 reprezentat pe un octet).

Tipurile posibile de împărțiri sunt aceleași ca la instrucțiunea DIV, cu observația că operanzii trebuie pregătiți folosind instrucțiunile CBW și CWD. Considerăm aceleași definiții de date ca la instrucțiunea DIV:

```
.code
; impartire octet la octet
      MOV   AL, B1
      CBW
      IDIV  B2                          ; AL = cat, AH = rest
;
; impartire cuvant la octet
      MOV   AX, W1
      IDIV  B1                          ; AL = cat, AH = rest
;
; impartire dublu-cuvant la cuvant
      MOV   AX, word ptr D1
      MOV   DX, word ptr D1 + 2
      IDIV  W1                          ; AX = cat, DX = rest
;
; impartire cuvant la cuvant
      MOV   AX, W1
      CWD
      IDIV  W2                          ; AX = cat, DX = rest
;
; impartire dublu-cuvant la byte
      MOV   AL, B1
      CBW
      MOV   BX, AX                      ; BX = Deimpartit
      MOV   AX, word ptr D1
```

```
        MOV   DX, word ptr D1 + 2      ; DX:AX = impartitor
        IDIV  BX                       ; AX = cat, DX = rest
```

O aplicație tipică a instrucțiunilor de împărțire este conversia unui cuvânt de 16 biți, cu sau fără semn, la un șir de caractere (cifre) în baza 10. Dacă presupunem că **n** este variabila care conține numărul, iar **sir** este variabila unde se depun cifrele generate, algoritmul poate fi descris (într-o notație specifică limbajului C), în felul următor:

```
adrsir = sir;
do {
      rest = n % 10;
      n = n/10;
      *sir++ = rest + '0';
} while (n != 0);
*sir = 0;
inverseaza(adrsir);
```

Interpretăm numărul n fără semn. În bucla de program cu test la partea inferioară, se calculează restul și câtul împărțirii lui n la 10. Câtul furnizează cifra curentă, căreia i se adună codul ASCII al cifrei 0. Se avansează adresa **sir** la următorul caracter, bucla terminându-se când n devine 0.

Se observă că acest algoritm furnizează cifrele numărului în ordine inversă; ca atare, rutina **inverseaza(sir)**, trebuie să inverseze ordinea caracterelor din șirul generat. De exemplu, dacă n = 1234, atunci prima iterație produce rest = 4 și n = 123, a doua iterație produce rest = 3 și n = 12 etc.

Să implementăm acest algoritm printr-o secvență în limbaj de asamblare.

```
.data
      n     dw ?
      sir   db 20 dup (?)
.code
      lea   si, sir
      mov   ax, n
      mov   dx, 0
      mov   bl, 10
      mov   bh, 0
_do:
      div   bx            ; AX = cat, DX (DL) = rest
      add   dl, '0'
      mov   [si], dl
      inc   si
      mov   dx, 0
      cmp   ax, 0
      jne   _do           ; Salt daca AX diferit de 0
      mov   byte ptr [si], 0
;
; Inverseaza
;
      mov   di, si
```

```
        dec   di            ; DI indica ultimul caracter util
        lea   si, sir       ; SI indica primul caracter
_test:
        cmp   si, di
        jae   _gata         ; Salt daca SI >= DI
        mov   al, [si]      ; Interschimba octetii de la
        xchg  al, [di]      ; adresele din SI si DI
        mov   [si], al
        inc   si
        dec   di
        jmp   _test
_gata:
```

Declarația **sir db 20 dup (?)** rezervă o zonă de 20 de octeți în segmentul de date. Simbolurile **_do:, _test:, _gata:** sunt etichete folosite la instrucțiuni de salt.

Deși avem de făcut o împărțire la 10 (deci la o cantitate care se poate reprezenta pe un octet), iar deîmpărțitul este un cuvânt, vom folosi totuși împărțirea dublu-cuvânt la cuvânt, pentru a evita depășirile. Se pregătește deci deîmpărțitul în DX:AX și câtul în BX. Pentru accesul la șirul de caractere, folosim registrul SI. Așadar, variabilele **n** și **sir** din algoritm vor fi ținute în registrele AX și SI.

Prin împărțire, se obține câtul în AX și restul în DX. De fapt, fiind o împărțire la 10, restul nu poate depăși valoarea 9, deci îl vom găsi, de fapt, în DL. Adăugăm '0' și depunem caracterul la adresa dată de SI, incrementăm SI și pregătim deîmpărțitul pentru pasul următor (adică forțăm DX = 0). Se testează acum AX (adică variabila n), prin comparație cu 0. Instrucțiunea JNZ înseamnă „Jump on Not Equal", deci salt dacă operanzii din ultima comparație sunt diferiți. Acest salt condiționat reia bucla de program.

Pentru a inversa caracterele generate, se poziționează registrele SI și DI pe primul și, respectiv, pe ultimul caracter util din șir, după care se execută o buclă de inversare, care continuă cât timp SI < DI. Instrucțiunea JAE înseamnă „Jump if Above or Equal", deci salt dacă la ultima comparație primul operand a fost mai mare sau egal cu al doilea. În bucla respectivă, se interschimbă caracterele de la adresele DI și SI, prin două MOV-uri și un XCHG.

Instrucțiunea AAD (ASCII Adjust for Division - Corecție ASCII înainte de împărțire)

Instrucțiunea nu are operanzi și efectuează o corecție a acumulatorului AX, interpretat ca două cifre BCD despachetate. Semnificația este următoarea:

```
(AL) ←  (AH) * 10 + (AL)
(AH) ←  0
```

Flaguri afectate: PF, SF și ZF, restul nedefinite.

Operația de corecție trebuie făcută înainte de împărțirea unui număr pe două cifre BCD, reprezentat pe un cuvânt, la o cifră BCD reprezentată pe un octet.

De exemplu, dacă AX = 0305H, adică valoarea BCD 35 și BL = 2, secvența de împărțire va fi:

```
AAD                 ; AX = 23H
DIV BL              ; AL = 11H, AH = 1
```

Instrucțiunea AAD încarcă AX cu valoarea 23H (35) și împărțirea se face corect, obținându-se câtul 11H (17) și restul 1.

2.2.6 Instrucțiuni logice (NOT, AND, TEST, OR, XOR)

Instrucțiunile logice realizează funcțiile logice de bază, pe octet sau pe cuvânt. Operațiile se fac la nivel de bit, deci se aplică funcția logică respectivă tuturor biților sau perechilor de biți corespunzători din operanzi. Instrucțiunea NOT are un singur operand, celelalte având fiecare doi operanzi.

Instrucțiunea NOT (Not - Negare logică bit cu bit)

Forma generală este:

```
NOT   destinatie
```

în care destinatie poate fi un registru sau o locație de memorie de 8 sau 16 biți. Instrucțiunea provoacă negarea tuturor biților operandului, deci calculul complementului față de 1.

Flaguri afectate: nici unul.

Instrucțiunea AND (And - Și logic bit cu bit)

Are forma generală:

```
AND   destinatie, sursa
```

în care destinație poate fi un registru sau o locație de memorie de 8 sau 16 biți, iar sursă poate fi un registru, o locație de memorie sau o constantă, pe 8 sau 16 biți. Semnificația este:

```
(destinatie) ←(destinatie) AND (sursa)
```

Flaguri afectate: SF, ZF, PF, CF = 0, OF = 0, AF nedefinit.

Instrucțiunea TEST (Test - Testează)

Are forma generală:

```
TEST  destinatie, sursa
```

în care destinatie și sursa sunt la fel ca la instrucțiunea AND. Efectul este execuția unei operații AND între cei doi operanzi, fără a se modifica destinația, dar cu poziționarea flagurilor la fel ca la instrucțiunea AND.

Flaguri afectate: SF, ZF, PF, CF = 0, OF = 0, AF nedefinit.

Această instrucțiune și raportul ei cu instrucțiunea AND sunt similare cu instrucțiunea CMP și raportul cu instrucțiunea SUB.

Instrucțiunea OR (Or - Sau logic bit cu bit)

Are forma generală:

```
OR    destinatie, sursa
```

în care destinatie și sursa sunt la fel ca la instrucțiunea AND. Semnificația este:

```
(destinatie) ←(destinatie) OR (sursa)
```

Flaguri afectate: SF, ZF, PF, CF = 0, OF = 0, AF nedefinit.

Instrucțiunea XOR (Exclusive Or - Sau-exclusiv bit cu bit)

Are forma generală:

```
XOR   destinatie, sursa
```

în care destinatie și sursa sunt la fel ca la instrucțiunea AND. Semnificația este:

```
(destinatie) ←(destinatie) XOR (sursa)
```

Flaguri afectate: SF, ZF, PF, CF = 0, OF = 0, AF nedefinit.

Funcția sau-exclusiv este 1 când operanzii săi sunt unul 0, iar celălalt 1 și este 0 când operanzii sunt ambii 0 sau ambii 1. Din acest motiv, funcția sau-exclusiv se mai numește și anticoincidență.

Instrucțiunile logice sunt folosite frecvent pentru anumite operații tipice, cum ar fi:

- ștergerea rapidă a unui registru, cu poziționarea flagurilor

```
XOR   AX, AX
XOR   CL, CL
```

- forțarea unor biți la valoarea 1, restul rămânând neschimbați

```
MASCA EQU  01101101B
OR    AL, MASCA
```

Pseudo-instrucțiunea EQU definește constante simbolice, iar prefixul B înseamnă număr scris în baza 2. În secvența de mai sus, biții cu valoarea 1 din MASCA vor fi forțați la 1 în registrul AL, iar biții cu valoarea 0 din MASCA vor rămâne neschimbați.

- forțarea unor biți la valoarea 0 cu ceilalți neschimbați

```
AND   AL, MASCA
```

Biții cu valoarea 0 din MASCA vor deveni 0 în AL, iar cei cu valoarea 1 în MASCA vor rămâne neschimbați.

- testarea unui singur bit dintr-un operand

```
TEST  AL, 01000000B
JZ    eticheta                    ; sau JNZ
```

- poziționarea flagurilor fără a modifica operandul

```
OR    AX, AX                      ; Pozitioneaza FLAGS
                                  ; conform AX
AND   AX, AX                      ; Similar
TEST  AX, AX                      ; Similar
```

- complementarea unui grup de biți, cu ceilalți neschimbați

Presupunem constanta MASCA, definită ca mai sus, și operandul aflat în AL. Dorim ca biții cu valoarea 1 din MASCA să fie complementați în AL, iar ceilalți să rămână neschimbați.

```
MASCA EQU 01110110B
MOV   BL, AL                      ; Salvare
AND   AL, MASCA                   ; Selectie biti care
                                  ; se modifica
NOT   AL                          ; Complementare
XCHG  AL, BL                      ; Refacere
AND   AL, NOT MASCA               ; Selectie biti care
                                  ; nu se modifica
OR    AL, BL                      ; Rezultat final
```

Expresia NOT MASCA este evaluată la asamblare, producându-se o constantă cu toți biții negați.

2.2.7 Instrucțiuni de deplasare (SHL, SAL, SHR, SAR) și de rotație (ROL, RCL, ROR, RCR)

Acest grup de instrucțiuni realizează operații de deplasare și de rotație la nivel de bit. Instrucțiunile au doi operanzi: primul este operandul propriu-zis, iar al doilea este numărul de biți cu care se deplasează sau se rotește primul operand. Ambele operații se pot face la dreapta sau la stânga. Deplasare înseamnă translatarea tuturor biților din operand la stânga/dreapta, cu completarea unei valori fixe în dreapta/stânga și cu pierderea biților din stânga/dreapta. Deplasarea cu un bit la stânga este echivalentă cu înmulțirea operandului cu 2, iar deplasarea la dreapta, cu împărțirea operandului la 2.

Rotație înseamnă translatarea tuturor biților din operand la stânga/dreapta, cu completarea în dreapta/stânga cu biții care se pierd în partea opusă.

Ambele operații se fac cu modificarea bistabilului CF, care poate chiar participa la operațiile de rotație.

Forma generală a instrucțiunilor este:

```
OPERATIE    OPERAND, CONTOR
```

în care OPERAND este un registru sau o locație de memorie de 8 sau 16 biți, iar CONTOR (numărul de biți) este fie constanta 1, fie registrul CL, care conține numărul de biți cu care se deplasează/rotește operandul. Procesoarele de generație nouă (80286 și peste) acceptă un număr oarecare de biți, specificat și printr-o constantă întreagă.

Flagurile sunt afectate în felul următor. La operațiile de deplasare, se modifică toate flagurile conform rezultatului, în afară de AF, care este nedefinit. La operațiile de rotație, se modifică numai CF și OF.

Modificarea flagului OF se face printr-un algoritm destul de complicat. Pentru a nu repeta acest algoritm, considerăm următoarele secvențe definite în pseudo-cod:

```
pozit_OF_left {
      daca CONTOR = 1, atunci
            daca b.c.m.s. din OPERAND este diferit de CF, atunci
                  OF ← 1
            altfel
                  OF ← 0
      altfel
            OF nedefinit
}

pozit_OF_right {
      daca CONTOR = 1, atunci
            daca cei doi biti mai semnif. din OPERAND sunt diferiti
                  OF ← 1
            altfel
                  OF ← 0
      altfel
            OF nedefinit
}
```

Se vede deci că flagul OF este poziționat numai dacă se face o operație de deplasare/rotație de 1 bit.

Descrierea instrucțiunilor se va face într-un format de tip pseudo-cod, cu evidențierea operațiilor aritmetice echivalente.

La instrucțiunile de deplasare, se consideră deplasări logice și aritmetice, care se pot utiliza după natura operanzilor.

Instrucțiunea SHL/SAL (Shift Logic/Arithmetic Left - Deplasează logic/aritmetic la stânga)

Are forma generală:

```
SHL/SAL     OPERAND, CONTOR
```

Deși există două mnemonice (SHL și SAL), în fapt este vorba de o unică instrucțiune. Semnificația este următoarea:

```
temp ← CONTOR
cat timp temp != 0 {
      CF ← b.c.m.s din OPR
      OPR ← 2*OPR (operatie fara semn)
      temp ← temp - 1
}
pozit_OF_left
```

Această descriere nu spune altceva decât că bitul cel mai semnificativ al operandului trece în CF, după care toți biții se deplasează la stânga cu o poziție (vezi Figura 2.3). Operația se repetă de atâtea ori cât este valoarea lui CONTOR (1 sau conținutul registrului CL).

Instrucțiunea SHR (Shift Logic Right - Deplasează logic la dreapta)

Are forma generală:

```
SHR   OPERAND, CONTOR
```

Semnificația este următoarea:

```
temp ← CONTOR
cat timp temp != 0 {
      CF ← b.c.m.p.s din OPR
      OPR ← OPR/2 (operatie fara semn)
      temp ← temp - 1
}
pozit_OF_right
```

Descrierea de mai sus spune că bitul cel mai puțin semnificativ din OPERAND trece în CF, după care se deplasează toți biții cu o poziție la dreapta (împărțire la 2). Faptul că operația de împărțire se execută fără semn înseamnă că se completează cu un bit 0 dinspre stânga (vezi Figura 2.3). Operația se repetă de atâtea ori cât este valoarea lui CONTOR (1 sau conținutul registrului CL).

Instrucțiunea SAR (Shift Arithmetic Right - Deplasează aritmetic la dreapta)

Are forma generală:

```
SAR   OPERAND, CONTOR
```

Semnificația este următoarea:

```
temp ← CONTOR
cat timp temp != 0 {
      CF ← b.c.m.p.s din OPR
      OPR ← OPR/2 (operatie cu semn)
      temp ← temp - 1
}
pozit_OF_right
```

Singura diferență față de deplasarea logică la dreapta este realizarea împărțirii la 2, luând în considerare semnul operandului. Aceasta înseamnă că se conservă bitul de semn, mai precis, completarea dinspre stânga se face cu bitul de semn (vezi Figura 2.3).

Aceste două tipuri de deplasare la dreapta se reflectă și în implementările limbajelor de nivel înalt. De exemplu, standardul ANSI al limbajului C lasă neprecizat faptul că deplasarea la dreapta a unui întreg de tip signed se face cu completare dinspre stânga cu 0 sau cu bitul de semn (deci operație de tip SHR sau SAR); această alegere revine implementării. Totuși, la deplasarea cantităților de tip unsigned, se completează întotdeauna dinspre stânga cu 0 (deci se face deplasare logică). Acest fapt este important mai ales în cazul tipului char, care nu este specificat în standardul ANSI ca fiind cu semn sau fără semn.

Instrucțiunea ROL (Rotate Left - Roteste la stânga)

Are forma generală:

```
ROL   OPERAND, CONTOR
```

Semnificația este următoarea:

```
temp ← CONTOR
cat timp temp != 0 {
      CF ← b.c.m.s. din OPERAND
      OPR ← 2*OPERAND + CF
      temp ← temp - 1
}
pozit_OF_left
```

Descrierea de mai sus spune că bitul cel mai semnificativ din OPERAND trece atât în CF, cât și în bitul cel mai puțin semnificativ din OPERAND, după ce toți biții acestuia s-au deplasat la stânga cu o poziție (vezi Figura 2.3). Operația se repetă de atâtea ori cât este valoarea lui CONTOR (1 sau conținutul registrului CL).

Instrucțiunea RCL (Rotate Left through Carry - Rotește la stânga prin carry)

Are forma generală:

```
RCL    OPERAND, CONTOR
```

Semnificația este următoarea:

```
temp ← CONTOR
cat timp temp != 0 {
        temp_cf ← CF
        CF ← b.c.m.s. din OPERAND
        OPR ← 2*OPERAND + temp_cf
        temp ← temp - 1
}
pozit_OF_left
```

Descrierea de mai sus spune că bitul cel mai semnificativ din OPERAND trece în CF, se deplasează toți biții din OPERAND cu o poziție la stânga, iar CF original trece în bitul cel mai puțin semnificativ din OPERAND. Cu alte cuvinte, CF participă efectiv la rotație (vezi Figura 2.3). Operația se repetă de atâtea ori cât este valoarea lui CONTOR (1 sau conținutul registrului CL).

Instrucțiunea ROR (Rotate Right - Rotește la dreapta)

Are forma generală:

```
ROR    OPERAND, CONTOR
```

Semnificația este:

```
temp ← CONTOR
cat timp temp != 0 {
        CF ← b.c.m.p.s din OPERAND
        OPR ← OPR/2 (operaÆie fara semn)
        b.c.m.s. din OPERAND <--- CF
        temp ← temp - 1
}
pozit_OF_right
```

Descrierea de mai sus spune că bitul cel mai semnificativ din OPERAND trece atât în CF, cât și în bitul cel mai puțin semnificativ, după ce toți biții s-au deplasat la dreapta cu o poziție (vezi Figura 2.3). Operația se repetă de atâtea ori cât este valoarea lui CONTOR (1 sau conținutul registrului CL).

Instrucțiunea RCR (Rotate Right through Carry - Rotește la dreapta prin carry)

Are forma generală:

```
RCR   OPERAND, CONTOR
```

Semnificația este următoarea:

```
temp ← CONTOR
cat timp temp != 0 {
      temp_cf ← CF
      CF ← b.c.m.p.s. din OPERAND
      OPR ← OPR/2 (operatie fara semn)
      b.c.m.s. din OPERAND ← temp_cf
      temp ← temp - 1
}
pozit_OF_right
```

Descrierea de mai sus spune că bitul cel mai puțin semnificativ din OPERAND trece în CF, se deplasează toți biții din OPERAND cu o poziție la dreapta, iar CF original trece în bitul cel mai semnificativ din OPERAND. Cu alte cuvinte, CF participă efectiv la rotație (vezi Figura 2.3). Operația se repetă de atâtea ori cât este valoarea lui CONTOR (1 sau conținutul registrului CL).

Semnificațiile celor 7 instrucțiuni de deplasare și rotație sunt ilustrate în Figura 2.3.

Să considerăm câteva exemple de rotații și deplasări.

Înmulțirea/împărțirea cu puteri ale lui 2. Operațiile de asemenea tip se execută mult mai eficient prin deplasări la stânga/dreapta. Secvența:

```
MOV   CL, 4
MOV   AH, 0
SHL   AX, CL
```

realizează înmulțirea lui AL cu valoarea 16. Rezultatul se regăsește în AX. Secvența:

```
MOV   CL, 3
SAL   BX, CL
```

realizează împărțirea lui BX (considerat număr cu semn) la valoarea 8.

Similar, se realizează și înmulțiri cu valori care se pot exprima prin sume cu un număr redus de termeni de puteri ale lui 2. Secvența următoare realizează înmulțirea unei valori N presupuse inițial în AL cu valoarea 13, prin deplasări și adunări repetate. Rezultatul se regăsește în BX.

```
MOV   AH, 0
MOV   BX, AX                    ; Salvare N
MOV   DX, AX                    ; Salvare N
MOV   CL, 3
SHL   AX, 3                     ; AX <--- N * 8
ADD   BX, AX                    ; BX <--- N * 8 + N
```

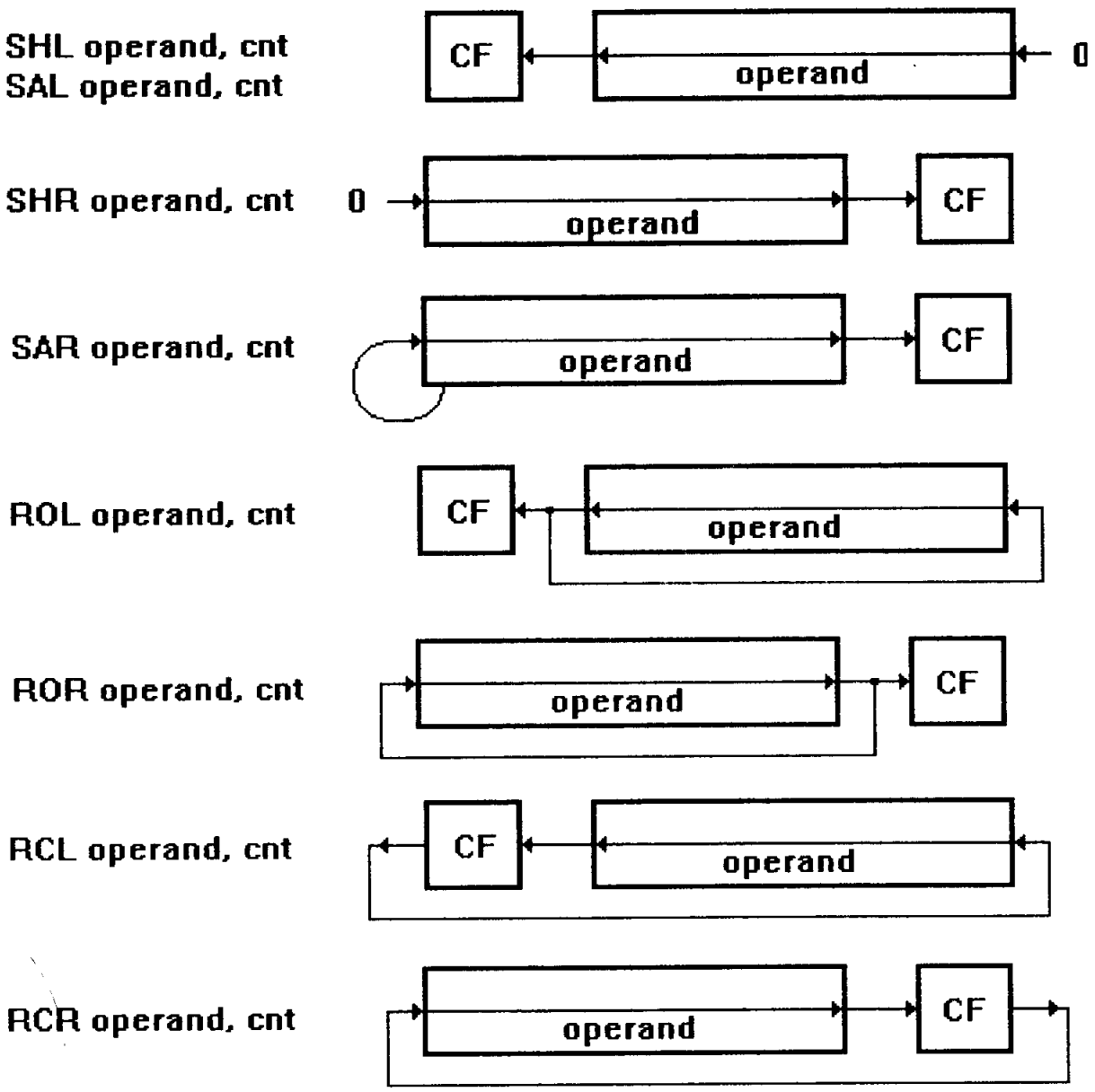


Figura 2.3 Semnificația operațiilor de deplasare și de rotație

```
MOV    AX, DX                    ; Refacere N
MOV    CL, 2
SHL    AX, CL                    ; AX <--- N * 4
ADD    BX, AX                    ; BX <--- N * 8 + N * 4 + N
```

2.3. Instrucțiuni pentru operații cu șiruri de caractere/cuvinte

Acest grup de instrucțiuni implementează un set puternic de operații cu șiruri de octeți sau de cuvinte. Prin șir se înțelege o secvență de octeți sau de cuvinte,

aflate la adrese succesive de memorie. Setul de instrucțiuni cuprinde operații primitive (la nivel de un octet sau un cuvânt) și prefixe de repetare, care pot realiza repetarea operațiilor primitive de un număr fix de ori sau până la îndeplinirea unei condiții de tip comparație.

2.3.1 Operații primitive

Operațiile primitive se grupează în patru categorii, fiecare putând fi pe octet sau pe cuvânt. Denumirea instrucțiunilor la nivel de octet se termină cu litera B, iar a celor la nivel de cuvânt cu litera W. Aceste instrucțiuni nu au operanzi.

- Move String (Copiază șir) MOVSB, MOVSW
- Compare String (Compară șiruri) CMPSB, CMPSW
- Load String (Încarcă șir în AL/AX) LODSB, LODSW
- Store String (Depune AL/AX în șir) STOSB, STOSW
- Scan String (Compară șir cu AL/AX) SCASB, SCASW

Toate operațiile primitive folosesc registrele DS:SI ca adresă sursă și/sau ES:DI ca adresă destinație. De asemenea, toate operațiile primitive actualizează adresele implicate în operație, în felul următor:

- dacă flagul DF = 0, adresele (registrul SI și/sau DI) sunt incrementate cu 1 sau cu 2, după cum operația primitivă este la nivel de octet sau de cuvânt;
- dacă flagul DF = 1, adresele (registrul și/sau DI) sunt decrementate cu 1 sau cu 2, după cum operația primitivă este la nivel de octet sau de cuvânt.

Vom utiliza notațiile:

```
(SI) ← (SI) + delta
(DI) ← (DI) + delta
```

unde delta este ±1 sau ±2, după starea bistabilului DF sau a tipului operației (octet sau cuvânt).

Starea flagului DF poate fi controlată prin instrucțiunile (fără operanzi) CLD (Clear Direction) și STD (Set Direction), care șterg, respectiv setează acest bistabil.

Instrucțiunile MOVSB, MOVSW

Semnificația este:

```
((ES:DI) ← ((DS:SI))
(SI) ← (SI) + delta
(DI) ← (DI) + delta
```

Se transferă deci un octet (MOVSB) sau un cuvânt (MOVSW) de la adresa dată de (DS:SI) la adresa dată de (ES:DI), după care se actualizează adresele.

Instrucțiunile CMPSB, CMPSW

Semnificația este:

```
((DS:SI) - ((ES:DI))
(SI) ← (SI) + delta
(DI) ← (DI) + delta
```

Se calculează temporar (fără a se modifica vreun operand) diferența dintre octeții (cuvintele) de la adresele (DS:SI) și (ES:DI), după care se actualizează adresele. Bistabilii de condiție se poziționează conform operației de scădere. Instrucțiunile se folosesc la testarea egalității/inegalității șirurilor.

Instrucțiunile LODSB, LODSW

Semnificația este:

```
(acumulator) ← ((DS:SI))
(SI) ← (SI) + delta
```

Se încarcă deci în AL, respectiv AX, octetul, respectiv cuvântul de la adresa (DS:SI), după care se actualizează această adresă.

Instrucțiunile STOSB, STOSW

Semnificația este:

```
((ES:DI)) ← (acumulator)
(DI) ← (DI) + delta
```

Se depune conținutul registrului AL, respectiv AX în octetul, respectiv cuvântul de la adresa (ES:DI), după care se actualizează această adresă.

Instrucțiunile SCASB, SCASW

Semnificația este:

```
(acumulator) - ((ES:DI))
(DI) ← (DI) + delta
```

Se calculează temporar (fără a se modifica vreun operand) diferența dintre registrul AL, respectiv AX și octetul, respectiv cuvântul de la adresa (ES:DI), după care se actualizează această adresă. Bistabilii de condiție se poziționează conform operației de scădere. Instrucțiunile se folosesc la testarea/căutarea unui anumit octet (cuvânt într-un șir).

Pe lângă formele fără operanzi, descrise mai sus, asamblorul mai recunoaște și forme în care operanzii apar explicit. Semnificația adreselor implicate în aceste instrucțiuni nu poate fi însă schimbată (se vor folosi tot registrele SI și DI). Singura rațiune pentru aceste forme este specificarea unui prefix de segment pentru adresa sursă. În acest caz, mnemonica instrucțiunii se scrie fără litera B

sau W de la sfârșit, dar este obligatorie specificarea tipului operației prin operatorul PTR. Nu se recomandă utilizarea acestor prefixe de segment din următoarele motive:

- nu se poate schimba registrul de segment al adresei destinație (acesta este tot timpul ES);
- în condițiile folosirii prefixelor de repetare (vezi 2.3.2), instrucțiunea rezultată ar avea atât prefix de repetare, cât și prefix de segment, ceea ce poate conduce la funcționări defectuoase, într-un context în care pot apărea întreruperi externe.

2.3.2 Prefixe de repetare (REP/REPE/REPZ, REPNE/REPNZ)

Prefixele de repetare permit execuția repetată a unei operații primitive cu șiruri de octeți sau de cuvinte, funcție de un contor de operații sau de un contor și o condiție logică. Aceste prefixe nu sunt instrucțiuni în sine, ci participă la formarea unor instrucțiuni compuse, alături de operațiile primitive descrise mai sus.

Prefixul de repetare REP/REPE/REPZ (Repeat - Repetă)

Cele trei mnemonice identifică de fapt un unic prefix de repetare. Forma generală a unei instrucțiuni cu acest prefix este:

```
REP/REPE/REPZ op_primitiva
```

Semnificația este:

```
cat timp CX != 0 {
      trateaza o eventuala intrerupere externa
      op_primitiva
      CX ←CX - 1
      daca op_primitiva este CMPS/SCAS çi ZF = 0, se iese din bucla
}
```

Se vede, deci, că operația primitivă se execută de un număr maxim de ori dat de conținutul registrului CX. Dacă CX este inițial 0, operația nu se execută nici o dată. În cazul operațiilor primitive CMPS și SCAS (care poziționează ZF), se iese forțat din buclă dacă ZF = 0, adică dacă rezultatul operației primitive este ne-nul. Bucla se execută deci cât timp acest rezultat este 0.

De obicei, scrierea cu REP se folosește la primitivele de tip MOVS, LODS și STOS, iar scrierea cu REPE sau REPZ la primitivele de tip CMPS și SCAS.

Să considerăm două exemple.

Secvența de mai jos transferă 100 de octeți de la adresa SURSA la adresa DESTINATIE, ambele presupuse în segmentul curent adresat prin DS.

```
.data
      SURSA db 100 dup (?)
      DEST  db 100 dup (?)
```

```
.code
        CLD             ; Directie ascendenta
        MOV   AX, DS    ; Pregatire
        MOV   ES, AX    ; Adrese
        LEA   SI, SURSA ; Adresa sursa
        LEA   DI, DEST  ; Adresa destinatie
        MOV   CX, 100   ; Contor
        REP   MOVSB     ; Instructiune compusa
```

Secvența următoare identifică primul octet diferit de un octet dat dintr-un șir de lungime 200 de octeți.

```
.data
      SIR   DB 200 dup (?)
.code
      MOV   AX, DS
      MOV   ES, AX
      LEA   DI, SIR
      CLD
      MOV   AL, 'A'       ; Se cauta primul octet
                          ; diferit de 'A'
      MOV   CX, 200       ; Numar maxim de iteratii
      REPE  SCASB         ; Bucla de cautare
```

Examinând bistabilul ZF la ieșirea din această secvență, putem deduce rezultatul căutării. Dacă ZF = 0 înseamnă că a avut loc o ieșire forțată din buclă, deci comparația a dat rezultatul 0. Registrul DI, decrementat cu o unitate, furnizează adresa primului octet diferit de octetul din AL. Dacă ZF = 1, înseamnă că toți octeții parcurși sunt identici cu octetul dat în AL.

La prima vedere, s-ar părea că, la fel de bine, am putea examina conținutul registrului CX la ieșirea din buclă (0 sau diferit de 0), pentru a deduce dacă octetul căutat a fost indentificat sau nu. Acest test este incorect, deoarece s-ar putea ca primul octet diferit de cel din AL să fie chiar ultimul. În acest caz, se iese din buclă cu CX = 0, la fel ca în situația în care toți octeții parcurși sunt identici cu cel din AL.

Prefixul de repetare REPNE/REPNZ (Repeat While Not Equal/Not Zero - Repetă cât timp diferit/diferit de zero)

Cele două mnemonice identifică un singur prefix de repetare. Forma generală este:

```
REPNE/REPNZ  operatie_primitiva
```

Semnificația este:

```
cat timp CX != 0 {
      trateaza o eventuala intrerupere externa
      op_primitiva
      CX ← CX - 1
      daca op_primitiva este CMPS/SCAS si ZF = 1, se iese din bucla
}
```

Diferența față de prefixul precedent este condiția de ieșire forțată din buclă: se iese dacă ZF = 1, deci dacă rezultatul operației primitive a fost 0. Ca atare, bucla se execută cât timp acest rezultat este ne-nul, dar nu mai mult de CX ori.

Practic, acest prefix de repetare se folosește numai cu operațiile primitive CMPS și SCAS. Pentru celelalte operații primitive, în care nu se ia în considerare bistabilul ZF, se preferă scrierea cu prefixul REP. La ieșirea din buclă, se poate examina bistabilul ZF, deducându-se dacă a fost sau nu o ieșire forțată.

Următoarea secvență determină ultimul caracter egal cu un caracter dat, prin parcurgerea șirului în sens invers.

```
.data
      SIR    db 30 (?)
.code
      LEA    DI, SIR
      ADD    DI, 29
      MOV    CX, 30
      STD
      MOV    AL, '?'        ; Se cauta primul octet = 'x', de
                            ; la dreapta la stanga
      REPNE SCASB
```

2.3.3 Operații complexe asupra șirurilor

Să considerăm unele operații complexe asupra șirurilor de caractere. Vom considera că șirurile au, ca ultim caracter, octetul 0, pe post de terminator de șir.

Compararea lexicografică a două șiruri

Acest algoritm primește ca date de intrare adresele a două șiruri de caractere terminate cu 0 și întoarce diferența primilor octeți care diferă în cele două șiruri sau 0 dacă cele două șiruri coincid. Următoarea secvență de program întoarce în AL rezultatul cerut. (Presupunem că DS și ES sunt inițializate corespunzător.)

```
.data
      SIR_1 DB 'Un exemplu de sir terminat cu zero', 0
      SIR_2 DB 'Un exemplu de sir terminat cu octetul nul', 0
.code
      CLD
      LEA    SI, SIR_1
      LEA    DI, SIR_2
IAR:
      LODSB                       ; (AL) = (SIR_1)
      TEST   AL, AL               ; Test terminator
      JZ     GATA
      SCASB                       ; Compara (AL) cu (SIR_2)
      JE     IAR                  ; Daca sunt egale, se reia bucla
      DEC    DI                   ; Daca nu, se revine pe caracterul
GATA:                             ; anterior
```

```
        SUB    AL, ES:[DI]          ; Se face diferenta
```

Copierea unui șir în alt șir

Următoarea secvență copiază un șir în alt șir. Copierea încetează după copierea terminatorului din șirul sursă. Se presupune că la adresa destinație se găsește suficient spațiu rezervat. Se presupune că registrele DS și ES indică segmentul curent de date.

```
.data
        SURSA DB 'Un sir care trebuie copiat',0
        DEST  DB 80 DUP (?)
.code
        LEA   SI, SURSA
        LEA   DI, DEST
        CLD
BUCLA:
        LODSB                      ; (AL) <--- (SURSA)
        STOSB                      ; (DEST) <--- (AL)
        TEST  AL,AL                ; Test terminator
        JNZ   BUCLA                ; Reluare
```

Calculul lungimii unui șir

Următoarea secvență determină în AX numărul de caractere utile din șirul SURSA. Terminatorul 0 nu se numără.

```
.code
        CLD
        LEA   DI, SURSA
        MOV   CX, 0FFFFH           ; Numarul maxim posibil
        MOV   BX, DI               ; Salvare adresa inceput
        XOR   AL, AL               ; Octet cautat
        REPNZ SCASB                ; Repeta cat timp e
                                   ; diferit de 0
        DEC   DI                   ; inapoi pe terminator
        SUB   DI, BX               ; Adresa terminator - Adresa de
        MOV   AX, DI               ; inceput = Lungime
```

Copierea a N caractere dintr-un șir în alt șir

Se copiază un număr de N caractere din șirul SURSA în șirul DEST. Dacă SURSA are mai puțin de N caractere, se completează DEST cu 0, până la N caractere.

```
.code
        CLD
        LEA   SI, SURSA
        LEA   DI, DEST
        MOV   CX, N
        OR    CX, CX
        JZ    GATA
```

```
BUCLA:
        LODSB                       ; Bucla de copiere
        STOSB                       ; pana la terminator
        DEC     CX                  ; inclusiv, dar nu mai
        JZ      GATA                ; mult de CX
        TEST    AL, AL              ; caractere
        JNZ     BUCLA
        REP     STOSB               ; Completare eventual cu 0
GATA:
```

Dacă N este 0, nu se copiază nimic. Se execută o buclă de copiere, în care se decrementează CX. Dacă CX = 0, totul se termină. Dacă s-a copiat terminatorul, se iese din bucla de copiere. În acest moment, AL = 0 și, dacă CX > 0, se execută depunerea lui AL în DEST, de atâtea ori cât este valoarea curentă a lui CX.

2.4 Instrucțiuni de apel de procedură și de salt (CALL, RET, JMP)

Procedurile se definesc în textul sursă după șablonul:

```
nume_proc PROC [ FAR | NEAR ]
                .
                .
                .
        RET
nume_proc ENDP
```

unde nume_proc este numele procedurii, iar parametrii FAR sau NEAR (opționali) indică tipul procedurii.

Procedurile sunt de două tipuri: FAR și NEAR. O procedură FAR poate fi apelată și din alte segmente de cod decât cel în care este definită, pe când o procedură NEAR poate fi apelată numai din segmentul de cod în care este definită.

Dacă se omit parametrii FAR sau NEAR, tipul procedurii este dedus din directivele simplificate de definire a segmentelor (modelul de memorie folosit). De exemplu, modelul LARGE presupune că toate procedurile sunt implicit de tip FAR.

În mod corespunzător, există apeluri de tip FAR, respectiv NEAR, precum și instrucțiuni de revenire de tip FAR, respectiv NEAR.

Instrucțiunea RET (Return) provoacă revenirea în programul apelant. Putem scrie o instrucțiune Return explicită, în formele RETN (Return Near) sau RETF (Return Far), sau RET pur și simplu, caz în care tipul instrucțiunii Return este dedus din tipul procedurii (FAR sau NEAR).

2.4.1 Apelul procedurilor și revenirea din proceduri

Instrucțiunea CALL (Apel de procedură)

Poate avea una din formele:

- CALL nume_proc
- CALL FAR PTR nume_proc
- CALL NEAR PTR nume_proc

În primul caz, tipul apelului este dedus din tipul procedurii, iar în celelalte, este specificat explicit (FAR sau NEAR).

Tipul apelului trebuie să coincidă cu tipul procedurii și cu tipul instrucțiunilor Return din interiorul procedurii, altfel se ajunge la funcționări defectuoase ale programului.

Semnificația instrucțiunii CALL este următoarea:

```
CALL de tip NEAR
        (SP) ← (SP) - 2
        SS:((SP) + 1 : (SP)) ← (IP)
        (IP) ← offset-ul primei instructiuni din procedura
```

Descrierea de mai sus spune că se salvează în stivă contorul program curent, într-o manieră similară instrucțiunii PUSH (se decrementează registrul SP cu 2 și se înscrie conținutul lui IP în vârful stivei). Registrul IP conține totdeauna adresa instrucțiunii care urmează (în memorie), după instrucțiunea care se excută în mod curent. Practic, se salvează în stivă adresa instrucțiunii de după instrucțiunea CALL. Această adresă este numită adresă de revenire.

După această salvare, se încarcă în IP adresa (deplasamentul) primei instrucțiuni din procedură, ceea ce înseamnă un transfer al controlului către procedură.

Este important de reținut că, în momentul intrării în procedură, în vârful stivei există adresa de revenire. Această adresă nu trebuie modificată în nici un fel, în caz contrar, revenirea în programul apelant nemaifiind posibilă.

De observat că, în secvența de apel a procedurii, registrul CS nu se modifică, ceea ce înseamnă că se rămâne în același segment de cod.

```
CALL de tip FAR
        (SP) ← (SP) - 2
        SS:((SP) + 1 : (SP)) ← (CS)
        (SP) ← (SP) - 2
        SS:((SP) + 1 : (SP)) ← (IP)
        (CS) ← adresa de segment a primei instructiuni din
               procedura
        (IP) ← offset-ul primei instructiuni din procedura
```

Ceea ce este diferit față de apelul de tip NEAR este faptul că se salvează adresa completă de revenire (pe 32 de biți), prin plasarea în stivă atât a registrului IP, cât și a registrului CS. Similar, transferul controlului se face prin modificarea explicită a perechii de registre (CS:IP).

De observat că instrucțiunea CALL de tip FAR este una din puținele instrucțiuni care modifică explicit registrul CS.

Instrucțiunea RET (Return - Revenire din procedură)

Există Return de tip FAR și Return de tip NEAR. Formele posibile ale instrucțiunii sunt:

- RETN [N]
- RETF [N]
- RET [N]

în care parantezele drepte spun că N este o constantă întreagă opțională.

În cea de-a treia formă, tipul instrucțiunii (NEAR sau FAR) este dedus din tipul procedurii.

Semnificația este următoarea:

```
Return de tip NEAR
        (IP) ← SS:((SP) + 1 : (SP))
        (SP) ← (SP) + 2
        [ (SP) ← (SP) + N ]
```

Se observă că se reface registrul (IP), prin copierea vârfului stivei și incrementarea registrului SP cu 2. Dacă SP are aceeași valoare ca la intrarea în procedură și conținutul stivei nu a fost alterat între timp, atunci se copiază practic în IP adresa de revenire, ceea ce provoacă transferul controlului la instrucțiunea care urmează instrucțiunii CALL care a provocat apelul procedurii.

Dacă în formatul instrucțiunii RET există constanta opțională N, atunci se adună această constantă la registrul SP. Acest tip de Return se numește Return cu descărcarea stivei.

```
Return de tip FAR
        (IP) ← SS:((SP) + 1 : (SP))
        (SP) ← (SP) + 2
        (CS) ← SS:((SP) + 1 : (SP))
        (SP) ← (SP) + 2
        [ (SP) ← (SP) + N ]
```

Se reface din stivă perechea de registre (CS:IP), cu actualizarea registrului SP și, dacă este prezentă constanta N, se adună N la SP.

Este important de reținut că instrucțiunile CALL și RET sunt instrucțiuni pereche: ele salvează, respectiv refac adresa de revenire. Pentru ca mecanismul de apel/revenire să funcționeze corect, trebuie îndeplinite condițiile:

- tipul instrucțiunii CALL și tipul instrucțiunii RET trebuie să coincidă (FAR sau NEAR);
- registrul SP din momentul execuției instrucțiunii RET să aibă aceeași valoare ca la intrarea în procedură (să indice adresa de revenire);
- adresa de revenire salvată în stivă să nu fi fost alterată de către procedură.

Încălcarea unora din cele trei condiții de mai sus constituie o eroare frecventă de programare. În astfel de cazuri, funcționarea programului este compromisă, deoarece se plasează în CS și/sau IP o adresă incorectă, unde probabil nici nu există vreun program cu sens. Acest tip de eroare se numește „execuție de date", adică procesorul ajunge să execute nu instrucțiuni cu sens (definite într-un segment de cod), ci o zonă oarecare de memorie (date).

Recunoașterea acestei erori este destul de ușoară: se blochează tastatura, pe ecranul calculatorului apar tot felul de caractere ciudate, difuzorul începe să țiuie etc. Singura soluție este în acest caz un reset general al calculatorului.

Putem, deci, enunța o regulă de aur a programării în limbaj de asamblare: **Verificați controlul stivei !**

2.4.2 Instrucțiunea de salt (JMP)

Are forma generală:

```
JMP   tinta
```

în care tinta specifică adresa de salt (punctul în care se va da controlul). Specificarea țintei se poate face printr-o etichetă sau printr-o expresie (vezi 2.4.3). Etichetele au asociat un tip (NEAR sau FAR) și pot fi:

- un nume de procedură;
- o etichetă definită cu : (de tip NEAR);
- o etichetă definită cu directiva LABEL.

Exemple de etichete:

```
et1:
aici:
et3   LABEL FAR
gata  LABEL NEAR
```

Există trei tipuri de instrucțiuni JMP:

- de tip SHORT - adresa țintă este situată la o adresă în domeniul [-128, +127] față de adresa instrucțiunii JMP;
- de tip NEAR - adresa țintă este în același segment de cod cu instrucțiunea JMP;

- de tip FAR - adresa țintă poate fi în alt segment de cod față de instrucțiunea JMP.

Tipurile de salt pot fi explicitate prin operatorul PTR, într-una din formele:

```
JMP    SHORT PTR tinta
JMP    NEAR PTR tinta
JMP    FAR PTR tinta
```

sau se deduc din atributele expresiei care precizează ținta. În cazul etichetelor, tipul etichetei (FAR sau NEAR) furnizează tipul saltului.

Semnificația instrucțiunii JMP este:

```
JMP de tip SHORT
      (IP) ← (IP) + distanta dintre offset-ul curent si
             cel tinta
JMP de tip NEAR
      (IP) ← offset-ul adresei tinta
JMP de tip FAR
      (IP) ← offset-ul adresei tinta
      (CS) ← segmentul adresei tinta
```

Din punctul de vedere al programatorului, faptul că la saltul de tip SHORT se adună la IP o diferență este similar cu modificarea explicită a lui IP de la JMP de tip NEAR.

Ceea ce diferă este codificarea internă a celor două instrucțiuni:

- la salt de tip SHORT, codul instrucțiunii conține diferența dintre offset-ul curent și cel al țintei. Această diferență se memorează intern pe un octet, de unde restricția de domeniu [- 128, +127];
- la salt de tip NEAR, codul instrucțiunii conține explicit offset-ul țintei.

Se observă că salturile de tip NEAR și FAR sunt similare cu apelurile de procedură de tip NEAR sau FAR, fără însă a se salva adresa de revenire și identificând numele procedurii cu o etichetă.

2.4.3 Tipuri de salt/apel

Tipurile de instrucțiuni de salt și de apel specifică modul de determinare a adresei țintă, respectiv al adresei procedurii. Deoarece aceste tipuri sunt identice pentru instrucțiunile JMP și CALL, vor fi prezentate împreună. Prin instrucțiuni de salt înțelegem aici cele de tip NEAR sau FAR (salturile de tip SHORT sunt implicit directe). De asemenea, în cele ce urmează identificăm etichetele cu numele de proceduri.

JMP/CALL de tip direct

Operandul care apare în formatul instrucțiunii este o etichetă care identifică punctul în care se dă controlul (adresa țintă). Salturile/apelurile directe pot fi:

- **salturi/apeluri directe intrasegment (NEAR)** - eticheta este în același segment de cod cu instrucțiunea JMP/CALL;
- **salturi/apeluri intersegment (FAR)** - eticheta poate fi definită și în alt segment de cod decât cel care conține instrucțiunea JMP/CALL.

Exemplu:

```
.code
ALFA:
      .........
BETA  LABEL FAR
      .........
GAMMA PROC FAR
      ............
GAMMA ENDP
      ............
JMP/CALL    ALFA              ; NEAR
JMP/CALL    BETA              ; FAR implicit
JMP/CALL    FAR PTR GAMMA     ; FAR explicit
```

JMP/CALL de tip indirect

Operandul care apare în formatul instrucțiunii JMP/CALL reprezintă un cuvânt (sau un dublu-cuvânt) din memorie, care conține adresa NEAR (sau FAR) unde se va da controlul.

Salturile/apelurile indirecte pot fi:

- **Salt/apel indirect intrasegment (NEAR)**

Forma generală este:

```
JMP/CALL    expr
```

în care **expr** poate fi:

- a) un registru care conține offset-ul țintei;
- b) o variabilă de tip WORD care conține offset-ul țintei;
- c) o expresie cu indici reprezentând un cuvânt din memorie care conține offset-ul țintei;
- d) o referire anonimă la un cuvânt din memorie care conține offset-ul țintei.

Să considerăm câteva exemple:

```
.data
      W_ALFA       dw    N_ETIC
      W_TAB_PROC   dw    N_PROC_0
                   dw    N_PROC_1
                   dw    N_PROC_2
.code
N_ETIC:                          ; Eticheta NEAR
      ............
N_PROC_i PROC NEAR               ; i = 0, 1, 2
```

```
        ...........
N_PROC_i ENDP
        ...........
LEA             BX, N_PROC_1
JMP/CALL        BX                      ; Tipul a)
JMP/CALL        W_ALFA                  ; Tipul b)
JMP/CALL        W_TAB_PROC [SI]         ; Tipul c); SI = 0, 2, 4
LEA             BX, W_TAB_PROC
JMP/CALL        WORD PTR [BX] [SI]; Tipul d); SI = 0, 2, 4
LEA             BX, W_ALFA
JMP/CALL        WORD PTR [BX]           ; Tipul d)
```

Formele cu referiri anonime la memorie trebuie însoțite de operatorul WORD PTR. Ultimele două exemple de salt/apel (tipul d) pun în evidență două nivele de indirectare (vezi Figura 2.4): de la BX la variabila W_ALFA și de la W_ALFA la eticheta N_ETIC.

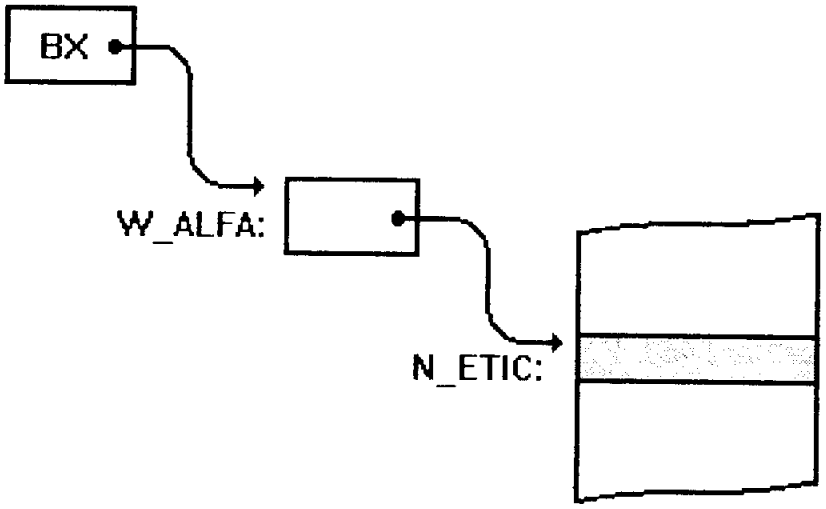


Figura 2.4 Cele două nivele de indirectare la CALL WORD PTR [BX]

- **Salt/apel indirect intersegment (FAR)**

Forma generală este:

```
JMP/CALL    expr
```

în care **expr** poate fi:

- a) o variabilă de tip DWORD care conține adresa completă a țintei;
- b) o expresie cu indici reprezentând un dublu cuvânt din memorie care conține adresa completă a țintei;
- c) o referire anonimă la un dublu cuvânt de memorie care conține adresa țintei.

Să considerăm câteva exemple:

```
.data
      D_ALFA       dd F_ETIC
      D_TAB_PROC   dd F_PROC_0
                   dd F_PROC_1
                   dd F_PROC_2
.code
F_ETIC LABEL FAR                    ; Eticheta FAR
      ............
F_PROC_i PROC FAR                         ; i = 0, 1, 2
      ............
F_PROC_i ENDP
      ............
JMP/CALL     D_ALFA                 ; Tipul a)
JMP/CALL     D_TAB_PROC [SI]        ; Tipul b); SI = 0, 4, 8
LEA          BX, D_TAB_PROC
JMP/CALL     DWORD PTR [BX][SI]; Tipul c); SI = 0, 4, 8
LEA          BX, D_ALFA
JMP/CALL     DWORD PTR [BX]         ; Tipul c)
```

Formele cu referiri anonime la memorie trebuie însoțite de operatorul DWORD PTR. Ultimele două exemple de salt/apel (tipul c) pun în evidență două nivele de indirectare (vezi Figura 2.5): de la BX la variabila DALFA și de la DALFA la eticheta ETIC.

Apelurile indirecte de proceduri trebuie să respecte tipul procedurii. Astfel, o instrucțiune de forma CALL WORD PTR [BX] trebuie folosită numai cu proceduri de tip NEAR, iar una de tip CALL DWORD PTR [BX] numai cu proceduri de tip FAR.

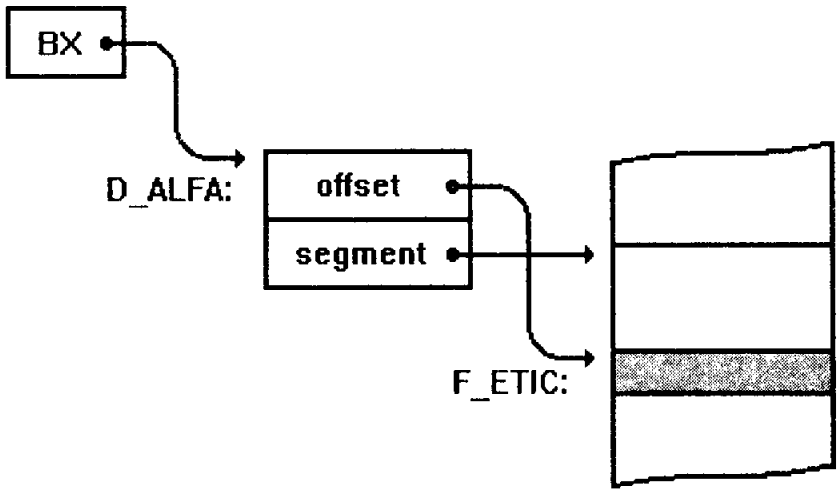


Figura 2.5 Cele două nivele de indirectare la CALL DWORD PTR [BX]

Să considerăm un exemplu de utilizare a tabelelor de adrese. Presupunem dezvoltarea unui meniu de comenzi, fiecare comandă fiind reprezentată de un

caracter alfabetic (de la 'A' la 'Z') Unele caractere pot să nu fie utilizate în meniu. De asemenea, presupunem caracterul în registrul AL și existența, pentru fiecare comandă, a unei proceduri de tratare.

O soluție evidentă, dar ineficientă este compararea lui AL cu toate caracterele meniului și cu apeluri corespunzătoare de proceduri, deci ceva de genul:

```
CMP    AL, 'A'
JZ     et_A
CMP    AL, 'B'
JZ     et_B ; etc.
```

Evident că, dacă meniul este mare, o asemenea secvență de program este greu de întreținut. Adăugarea sau eliminarea unei comenzi presupune parcurgerea cu atenție a codului sursă.

Soluția adecvată a problemei constă în definirea unei tabele cu adresele procedurilor de tratare, în ordinea alfabetică a comenzilor. Pentru literele de alfabet care nu sunt alocate nici unei comenzi, se scrie o procedură specială PROC_NULL. La execuție, se va calcula poziția respectivă din tabelă, chiar pe baza caracterului din AL și se va genera un apel indirect de procedură.

```
.data
      TAB_CMD      DW PROC_A
                   DW PROC_B
                   DW PROC_NULL       ; Comanda 'C' nu exista
                   DW PROC_D
                                      ;  etc.
                   DW PROC_Z
.code
      SUB    AL, 'A'                  ; AL = 0...27
      ADD    AL, AL                   ; AL <-- 2*AL
      ADD    AL, AL                   ; Se pune in cazul
                                      ; procedurilor FAR
      MOV    BL, AL
      XOR    BH, BH                   ; BX = intrare
                                      ; in tabela
      CALL   WORD PTR TAB_CMD[BX]
```

În cazul procedurilor far, se definește tabela TAB_CMD cu directiva Define DoubleWord și se calculează BX ← 4*AL, în loc de BX ← 2*AL.

Această implementare este foarte ușor de întreținut. Eliminarea unei comenzi se face punând adresa procedurii POC_NULL pe poziția corespunzătoare. Adăugarea unei comenzi se face înlocuind procedura PROC_NULL cu procedura de tratare a noii comenzi.

Nici un tip de instrucțiune de salt sau de apel de procedură nu modifică vreun bistabil de condiție.

2.4.4 Instrucțiuni de salt condiționat

Instrucțiunile din această categorie implementează salturi condiționate de valoarea unor bistabili de condiție. Prin extensie, instrucțiunile de salt obișnuite (JMP) se mai numesc și salturi necondiționate. Instrucțiunile de salt condiționat au următoarele caracteristici:

- toate instrucțiunile de salt condiționat sunt de tip SHORT (deci directe), ceea ce înseamnă că adresa țintă trebuie să fie la o distanță cuprinsă între -127 și +128 de octeți față de instrucțiunea de salt;
- există mai multe mnemonice pentru aceeași instrucțiune;
- dacă condiția nu este îndeplinită, saltul nu are loc, deci execuția continuă cu instrucțiunea următoare;
- există instrucțiuni atât pe condiția directă, cât și pe condiția negată;
- bistabilii de condiție nu sunt afectați.

În Tabelul 2.6, se prezintă instrucțiunile de salt condiționat. Prima coloană listează mnemonicele instrucțiunilor, coloana a doua listează condiția de salt, iar coloana a treia, interpretarea condiției respective. Primele opt linii reprezintă salturile pe condițiile directe, iar celelalte opt, salturile pe condițiile corespunzătoare negate.

Denumirile instrucțiunilor se citesc în forma:

Jump (Salt) on (pe) <Interpretare>

unde <Interpretare> este textul din coloana a treia.

Instrucțiunile de salt condiționat se utilizează după o instrucțiune care modifică bistabilii de condiție. Cea mai des întâlnită situație este instrucțiunea de comparație CMP.

Se observă că există două categorii de instrucțiuni pentru noțiunile de „mai mic" și „mai mare": cele care conțin cuvintele „above" sau „bellow" și cele care conțin cuvintele „less" sau „greater". Primele se folosesc în situația comparației a două cantități fără semn, iar ultimele, în situația comparației unor cantități cu semn.

Să considerăm următoarele secvențe de program, în care se compară valorile 0FFH și 1:

```
MOV   AL, 0FFH          MOV   AL, 0FFH
MOV   BL, 1             MOV   BL, 1
CMP   AL, BL            CMP   AL, BL
JA    ET_1              JG    ET_1
```

Se observă că (AL) > (BL) dacă interpretăm cele două valori fără semn (255 > 1) și că (AL) < (BL) dacă interpretăm cele două valori cu semn (-1 < 1). Astfel, în primul exemplu, saltul la eticheta ET_1 are loc, pe când în cel de-al doilea nu.

Condițiile asupra bistabililor în cele două tipuri de instrucțiuni și semnificația bistabililor respectivi (vezi 2.2.1) explică cele două tipuri de condiții.

Reținem deci următoarele reguli:

- La comparații cu semn, folosim „GREATER" și „LESS"
- La comparații fără semn, folosim „ABOVE" și „BELLOW"

Instrucțiune (mnemonică)	Condiție de salt	Interpretare
JE, JZ	ZF = 1	Zero, Equal
JL, JNGE	S F ≠ OF	Less, Not Greater or Equal
JLE, JNG	SF ≠ OF sau ZF = 1	Less or Equal, Not Greater
JB, JNAE, JC	CF = 1	Below, Not Above or Equal, Carry
JBE, JNA	CF = 1 sau ZF = 1	Below or Equal, Not Above
JP, JPE	PF = 1	Parity, Parity Even
JO	OF = 1	Overflow
JS	SF = 1	Sign
JNE, JNZ	ZF = 0	Not Zero, Not Equal
JNL, JGE	SF = OF	Not Less, Greater or Equal
JNLE, JG	SF = OF și ZF = 0	Not Less or Equal, Greater
JNB, JAE, JNC	CF = 0	Not Below, Above or Equal, Not Carry
JNBE, JA	CF = 0 și ZF = 0	Not Below or Equal, Above
JNP, JPO	PF = 0	Not Parity, Parity Odd
JNO	OF = 0	Not Overflow
JNS	SF = 0	Not Sign

Tabelul 2.6 Instrucțiunile de salt condiționat

Uneori este necesar să scriem instrucțiuni de salt condiționat la etichete care ies în afara domeniului [-128, +127] față de instrucțiunea curentă. În această situație, folosim următoarea schemă, prin care înlocuim saltul pe o condiție directă „departe" cu un salt pe condiția negată „aproape" și cu un salt necondiționat „departe".

```
        Salt conditionat  pe conditie directa la ET_1
        ; Eticheta ET_1 e prea departe
ET_1:
```

Forma echivalentă este:

```
        Salt conditionat pe conditie negata la ET_2
        Salt neconditionat la ET_1
ET_2:
```

De exemplu, instrucțiunea:

```
        JE      ET_1
```

se substituie cu:

```
        JNE     ET_2
        JMP     ET_1
ET_2:
```

În schema echivalentă, eticheta ET_1 poate fi la orice distanță față de punctul de salt.

2.4.5 Instrucțiuni pentru controlul buclelor de program (JCXZ, LOOP, LOOPZ/LOOPE, LOOPNZ/LOOPNE)

Instrucțiunea JCXZ (Jump if CX is Zero - Salt dacă CX este zero)

Are forma generală:

```
        JCXZ   etic
```

unde etic este o etichetă aflată în domeniul [-128, +127] față de instrucțiunea curentă (la fel ca la salturile de tip SHORT).

Semnificația este evidentă:

```
daca (CX) = 0
        (IP) ← (IP) + distanta dintre offset-ul curent si
        cel tinta
```

adică se sare la eticheta specificată dacă CX conține valoarea 0.

Instrucțiuni de ciclare (LOOPxx - Ciclează)

Aceste instrucțiuni sunt, de fapt, salturi condiționate de valoarea registrului CX și de bistabilul ZF. La fel ca prefixele de repetare, cu care, de altfel, se aseamănă oarecum, există câte două mnemonice pentru aceeași instrucțiune. Forma lor generală este:

```
        LOOPxx  etic
```

unde etic este o etichetă aflată în domeniul [-128, +127] față de instrucțiunea curentă.

Toate aceste instrucțiuni utilizează registrul CX ca număr de iterații.

Instrucțiunea LOOP

Semnificația este următoarea:

```
        CX ← CX - 1
        daca CX != 0, atunci
```

```
(IP) ← (IP) + distanta dintre offset-ul curent si
cel tinta
```

Cu alte cuvinte, se decrementează CX și, dacă acesta este diferit de zero, se sare la eticheta specificată.

Următoarea secvență calculează suma (pe 16 biți) a elementelor unui tablou TAB DE 100 de întregi și o depune în variabila SUMA.

```
.data
        TAB     DW 100 DUP (0)
        SUMA    DW ?
.code
        XOR     AX, AX                  ; Initializeaza suma
        MOV     CX, 100                 ; Numar de iteratii
        MOV     SI, AX                  ; Initializeaza indice
NEXT:
        ADD     AX, TAB [SI]            ; Calcul suma
        ADD     SI, 2                   ; Actualizare indice
        LOOP    NEXT                    ; Cicleaza
        MOV     SUMA, AX                ; Depune rezultat
```

Forma generală a unei bucle de program realizată cu instrucțiunea LOOP este:

```
start_bucla:
        ; ;
        ; Corp bucla ;
        ; ;
       LOOP  start_bucla
```

Corpul buclei se va executa de atâtea ori cât este conținutul inițial al registrului CX (presupunând că acesta nu este modificat explicit în interiorul buclei).

Trebuie observat că registrul CX este mai întâi decrementat și apoi testat. Ca atare, dacă (CX) este inițial 0, bucla de program se va executa de 65535 de ori. Protecția față de o asemenea situație se realizează prin instrucțiunea JCXZ. Forma corectă a buclei de program va fi:

```
JCXZ end_bucla
start_bucla:
        ; ;
        ; Corp bucla ;
        ; ;
      LOOP start_bucla
end_bucla:
```

În forma de mai sus, dacă (CX) este inițial 0, corpul buclei nu se execută niciodată.

În situația în care corpul buclei ocupă mai mult de 128 de octeți, trebuie să renunțăm la JCXZ și LOOP și să le înlocuim cu comparații și cu salturi explicite. Bucla de program va avea forma generală:

```
        TEST  CX, CX
        JNZ   start_bucla
        JMP   end_bucla
start_bucla:
        ; ;
        ; Corp bucla ;
        ; ;
        DEC   CX
        TEST  CX, CX
        JZ    end_bucla
        JMP   start_bucla
end_bucla:
```

Următorul exemplu calculează primele 20 de numere Fibonacci (fib(n), n =1,...,20), definite recursiv prin:

f(0) = 0, f(1) = 1

f(k) = f(k-1) + f(k-2), dacă n > 1

și le depune în tabloul FIBO. Cele trei valori (f(k), f(k-1) și f(k-2)) care intervin în calcul sunt ținute în registrele AX, BX și DX.

```
.data
        FIBO DW 20 DUP (?)
.code
        MOV BX, 0           ; f(0)
        MOV DX, 1           ; f(1)
        MOV CX, 20          ; Contor
        MOV DI, 0           ; Indice in tabloul
bucla:
        MOV AX, BX
        ADD AX, DX          ; f(k) <-- f(k-1) + f(k-2)
        MOV FIBO [DI], AX ; Depune rezultat
        ADD DI, 2           ; Actualizeaza indice
        MOV DX, BX          ; f(k-1) devine f(k-2)
        MOV BX, AX          ; f(k) devine f(k-1)
        LOOP bucla          ; Cicleaza
```

Instrucțiunea LOOPZ/LOOPE (Loop While Zero/Equal - Ciclează cât timp este zero/egal)

Semnificația este următoarea:

```
CX ← CX - 1
      daca CX != 0 si ZF = 1, atunci
            (IP) ← (IP) + distanta dintre offset-ul curent si
                   cel tinta
```

Se decrementează deci CX și, dacă acesta este diferit de zero și bistabilul ZF este 1 (adică rezultatul ultimei operații aritmetice a fost zero), se sare la eticheta specificată. De obicei, pentru poziționarea bistabilului ZF se utilizează o instrucțiune de comparație.

Exemplul următor determină primul întreg diferit de zero dintr-un tablou TABL de 100 de întregi.

```
.code
        MOV CX, 100
        LEA BX, TABL
        MOV SI, -2
next:
        ADD SI, 2
        CMP WORD PTR [BX][SI], 0
        LOOPZ next
        JNZ gasit
```

Dacă bistabilul ZF este 0 la ieșirea din buclă înseamnă că s-a identificat primul întreg diferit de 0, iar SI conține adresa acestui întreg. În caz contrar, toate cele 100 de elemente ale tabloului sunt nule.

Instrucțiunea LOOPNZ/LOOPNE (Loop While Not Zero/Not Equal - Ciclează cât timp este diferit de zero/diferit)

Semnificația este următoarea:

```
CX ← CX - 1
      daca CX != 0 si ZF = 0, atunci
 (IP) ← (IP) + distanta dintre offset-ul curent si cel
        tinta
```

Condiția asupra bistabilului ZF este negata condiției de la LOOPZ/LOOPE. Efectul este că se ciclează cât timp rezultatul ultimei operații aritmetice este diferit de zero, dar nu de mai multe ori cât este conținutul inițial al lui CX.

De remarcat asemănările și deosebirile care există între instrucțiunile de ciclare și prefixele de repetare de la operații cu șiruri (vezi 2.3.2).

2.5. Întreruperi

Noțiunea de întrerupere presupune (așa cum îi arată și numele) întreruperea programului în curs de execuție și transferul controlului la o anumită rutină specifică, numită rutină de tratare, dictată de cauza care a generat întreruperea. Mecanismul prin care se face acest transfer este, în esență, de tip apel de procedură, ceea ce înseamnă revenirea în programul întrerupt, după terminarea rutinei de tratare.

2.5.1 Întreruperi hardware și software. Tabela de întreruperi

Întreruperile se pot clasifica după mai multe criterii, rezultând tipurile ilustrate în Figura 2.7.

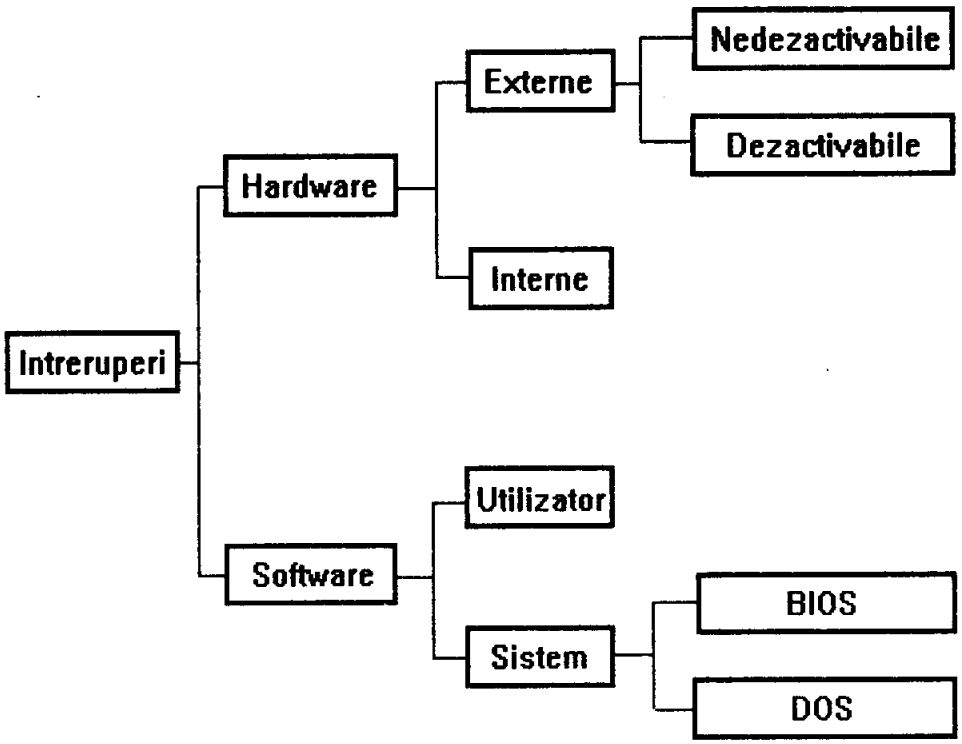


Figura 2.7 Clasificarea întreruperilor

Întreruperile software apar ca urmare a execuției unor instrucțiuni, cum ar fi INT, INTO, DIV, IDIV;

Întreruperile hardware externe sunt provocate de semnale electrice care se aplică pe intrările de întreruperi INT și NMI ale procesorului;

Întreruperile hardware interne apar ca urmare a unor condiții speciale de funcționare a procesorului (cum ar fi execuția pas cu pas a programelor);

Întreruperile dezactivabile sunt provocate de semnalul electric aplicat pe linia (intrarea) INT și sunt luate în considerare numai dacă bistabilul IF este 1.

Întreruperile nedezactivabile sunt provocate de semnalul electric aplicat pe linia NMI și sunt totdeauna luate în considerare.

Întreruperile hardware dezactivabile sunt controlate de unul sau mai multe circuite specializate (controlere de întreruperi 8259A), care acceptă fiecare cel mult opt cereri de întreruperi, pe care le transmit către linia INT a procesorului. Dacă bistabilul IF este 1, procesorul răspunde printr-o secvență de semnale electrice INTA (Interrupt Acknowledge), asemănătoare unei secvențe de citire a codului unei instrucțiuni. Pe durata unuia din aceste semnale, controlerul care a inițiat cererea de întrerupere plasează pe magistrala de date octetul care identifică nivelul întreruperii. Un asemenea sistem se numește sistem vectorizat de întreruperi (vezi Figura 2.8).

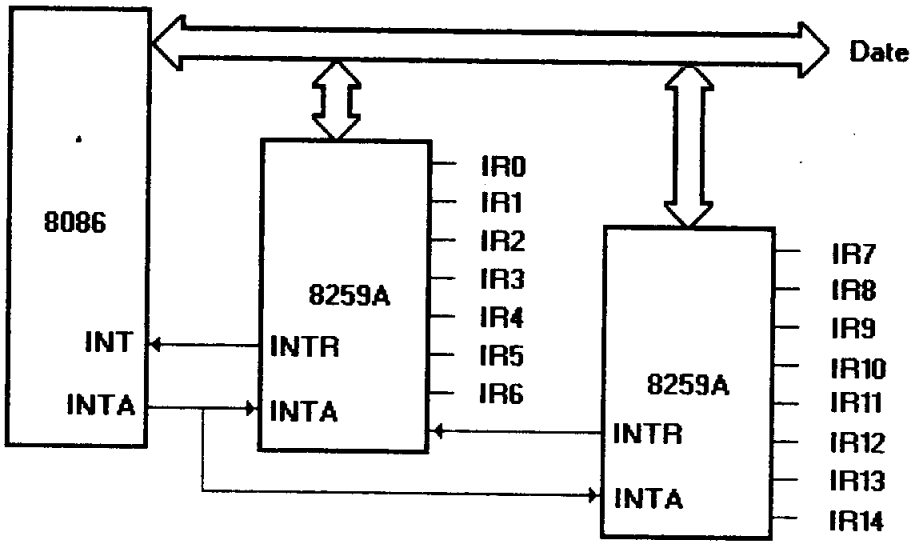


Figura 2.8 Întreruperi hardware vectorizate

Într-un sistem cu procesor 8086, pot exista maxim 256 de întreruperi (nivele) distincte. Fiecare din aceste nivele poate avea asociată o procedură de tip far, numită rutină de tratare. Adresele acestor rutine sunt trecute într-o așa numită tabelă de întreruperi, aflată la adresele fizice 00000÷003FFH, ocupând deci 1024 de octeți. Fiecare nivel ocupă 4 octeți, primii reprezentând offset-ul, iar următorii adresa de segment a procedurii (vezi Figura 2.9).

La apariția unei întreruperi, au loc următoarele acțiuni:

- se salvează în stivă registrele FLAGS, CS și IP (în această ordine);
- se șterg bistabilii IF și TF;
- se furnizează procesorului un întreg pe 8 biți (deci în gama 0÷255), numit și vector de întrerupere, care identifică nivelul asociat întreruperii;
- se execută un salt indirect intersegment la adresa de început a rutinei de tratare, prin intermediul tabelei de întreruperi.

Vectorul de întrerupere poate fi furnizat procesorului în următoarele moduri:

- în cazul întreruperilor hardware interne, nivelul este implicit; de exemplu, pentru întreruperea de execuție pas cu pas (generată dacă bistabilul TF=1), se utilizează totdeauna nivelul 1;
- în cazul întreruperilor hardware externe, nivelul este plasat pe magistrala de date, în cadrul ciclului mașină (Interrupt Acknowledge), care urmează după luarea în considerare a întreruperii de către dispozitivul care a generat întreruperea;
- în cazul întreruperilor software, nivelul este conținut în instrucțiune.

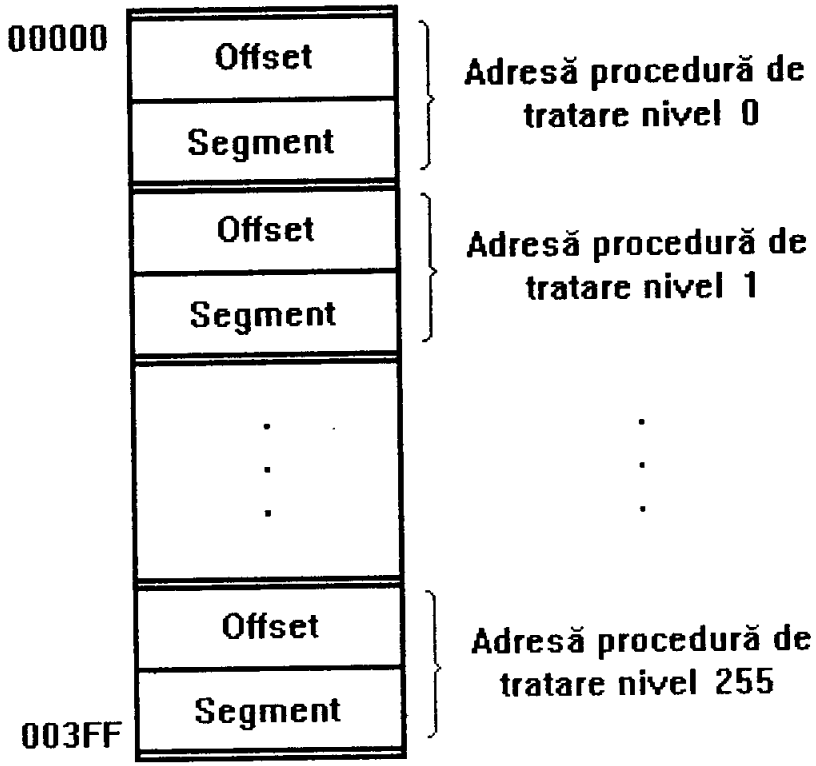


Figura 2.9 Tabela de întreruperi

Se observă că, exceptând salvarea flagurilor și ștergerea bistabililor IF și TF, secvența de tratare a unei întreruperi se reduce la un apel indirect de procedură far, prin intermediul tabelei de întreruperi.

2.5.2 Instrucțiuni specifice întreruperilor (INT, IRET, INTO)

Instrucțiunea INT (Interrupt - Întrerupere software)

Are forma generală:

```
INT n
```

unde n este o constantă întreagă în gama 0÷255.

Semnificația este următoarea:

```
(SP) ← (SP) - 2
SS:((SP) + 1 : (SP)) ← (FLAGS)
IF ← 0, TF ← 0
(SP) ← (SP) - 2
SS:((SP) + 1 : (SP)) ← (CS)
(CS) ← (4*n + 2)
(SP) ← (SP) - 2
```

```
SS:((SP) + 1 : (SP)) ← (IP)
(IP) ← (4*n)
```

Se încarcă, deci, în CS și IP conținutul de la adresele fizice 4•n+2 și 4•n, adică cei patru octeți corespunzători nivelului n din tabela de întreruperi.

Instrucțiunea IRET (Interrupt Return - Revenire din întrerupere)

Datorită acțiunilor care au loc la apariția unei întreruperi, o procedură de tratare nu se poate încheia cu o instrucțiune RETF, deoarece trebuie refăcut și registrul de flaguri. Ca atare, procedurile de tratare a întreruperilor se încheie totdeauna cu instrucțiunea IRET.

Instrucțiunea, care este fără operanzi, are semnificația următoare:

```
(IP) ← SS:((SP) + 1 : (SP))
(SP) ← (SP) + 2
(CS) ← SS:((SP) + 1 : (SP))
(SP) ← (SP) + 2
(FLAGS) ← SS:((SP) + 1 : (SP))
(SP) ← (SP) + 2
```

Se observă că bistabilii IF și TF, care au fost șterși la apariția întreruperii, sunt refăcuți așa cum erau înainte de întrerupere, o dată cu ceilalți bistabili din registrul FLAGS.

Instrucțiunea INTO (Interrupt if Overflow - Întrerupere în caz de depășire)

Instrucțiunea, care este fără parametri, are semnificația următoare:

- dacă OF = 1, atunci se execută secvența corespunzătoare unei instrucțiuni INT 4.

Codificarea instrucțiunilor INT este pe doi octeți, cu excepția instrucțiunii INT 3, la care codificarea este pe un singur octet. Acest fapt nu este întâmplător, deoarece nivelele 2 și 3 sunt, în general, utilizate de programele de depanare (debuggere).

Astfel, rularea pas cu pas a unui program se face setând bistabilul TF și scriind pe nivelul 1 al tabelei de întreruperi adresa unei rutine proprii de tratare. La execuția fiecărei instrucțiuni, se va da controlul rutinei proprii, care va afișa corespunzător starea programului. Rularea până la o adresă dată (breakpoint) se realizează prin înlocuirea octetului de la adresa respectivă cu codul unei instrucțiuni INT 3. Astfel, când programul care se execută ajunge la acea adresă, se dă controlul rutinei de tratare pe nivelul 3.

Nivelele predefinite de întrerupere sunt, deci:

- 0 - depășire la împărțire (cauze posibile: instrucțiunile DIV sau IDIV);

- 1 - execuție pas cu pas (cauză posibilă: bistabilul TF = 1);
- 2 - întrerupere externă nedezactivabilă (cauză posibilă: semnal electric pe linia de întrerupere nedezactivabilă NMI);
- 3 - execuție pas cu pas (cauză posibilă: instrucțiunea INT 3);
- 4 - depășire (cauză posibilă: instrucțiunea INTO).

La calculatoarele de tip IBM-PC, se mai pot cita întreruperile hardware de la ceasul de timp real (nivelul 8) și de la tastatură (nivelul 9).

Întreruperile software în gama 20H÷2FH sunt folosite de sistemul de operare DOS, iar cele în gama 10H÷1AH de către subsistemul de intrări-ieșiri BIOS.

2.6 Instrucțiuni pentru controlul procesorului

Toate instrucțiunile de acest gen sunt fără operanzi.

O primă categorie de instrucțiuni se referă la controlul explicit al unor bistabili de condiție (CLC, STC, CMC, CLD, STD, CLI și STI).

Instrucțiunile **CLC (Clear Carry Flag)**, **STC (Set Carry Flag)** și **CMC (Complement Carry Flag)** efectuează operațiile de ștergere, setare și, respectiv, complementare a bistabilului CF.

Instrucțiunile **CLD (Clear Direction Flag)** și **STD (Set Direction Flag)** șterg, respectiv, setează bistabilul DF.

Instrucțiunile **CLI (Clear Intrerrupt Flag)** și **STI (Set Interrupt Flag)** șterg, respectiv, setează bistabilul IF. Când IF = 0, întreruperile externe nedezactivabile nu sunt luate în considerare. Astfel, o așa numită secvență critică de program (care nu dorim să fie întreruptă) se protejează printr-o instrucțiune CLI înainte și o instrucțiune STI după. O secvență critică tipică este modificarea explicită a tabelei de întreruperi.

O altă serie de instrucțiuni (HALT, LOCK, WAIT) realizează unele operații speciale asupra procesorului.

Astfel, instrucțiunea **HALT (Oprire procesor)** forțează procesorul într-o stare specială de inactivitate, din care se poate ieși doar prin întreruperi externe sau prin reset general.

Prefixul **LOCK (Blocare magistrală)** se poate folosi înaintea oricărei instrucțiuni, efectul fiind că pe durata instrucțiunii, accesul unui alt dispozitiv la magistrala sistemului hardware este blocat. Într-un sistem multiprocesor, nu este suficient să protejăm secvențele critice prin instrucțiuni CLI/STI, deoarece controlul magistralei ar putea fi preluat de un alt procesor. De aici, utilitatea prefixului LOCK.

Instrucțiunea **WAIT (Așteaptă)** este folosită pentru a sincroniza activitatea procesorului cu cea a unui alt dispozitiv, de obicei un coprocesor matematic.

Execuția unei instrucțiuni matematice este preluată de coprocesor; totuși, procesorul de bază nu poate continua programul până nu primește un semnal de la coprocesor, prin care se anunță sfârșitul operației executate. Această sincronizare se realizează printr-un semnal electric aplicat pe linia TEST a procesorului de bază. Ca atare, instrucțiunea WAIT va forța procesorul de bază într-o stare de inactivitate, până la apariția unui semnal electric pe linia TEST.

În fine, există și o instrucțiune care nu face nimic: instrucțiunea **NOP (No Operation)**. Scopul unei asemenea instrucțiuni este de a introduce o întârziere în program. Asamblorul recunoaște mnemonica NOP, dar codificarea internă este aceeași cu a instrucțiunii XCHG AX, AX, care, evident, nu face nimic.

2.7 Dezvoltarea programelor în limbaj de asamblare

Acum, după ce am parcurs setul de instrucțiuni al procesorului, putem trece la prezentarea unui cadru de dezvoltare al programelor în limbaj de asamblare. Contextul ales este cel al calculatorului de tip IBM-PC sub sistemul de operare DOS și al produselor software Borland: asamblorul **TASM** (Turbo Assembler), link-editorul **TLINK** (Turbo Linker), bibliotecarul **TLIB** (Turbo Librarian) și depanatorul interactiv **TD** (Turbo Debugger).

Vom utiliza extensiile implicite ale fișierelor: **.ASM** pentru fișiere sursă, **.OBJ** pentru fișiere obiect, **.EXE** sau **.COM** pentru fișiere executabile.

O problemă specifică limbajului de asamblare este absența unor instrucțiuni de intrare-ieșire de nivel înalt. Pentru asemenea operații, va trebui să ne scriem propriile rutine. În această lucrare, vom utiliza o serie de proceduri și macroinstrucțiuni, implementate în fișierul **IO.H** (fișier header, listat în Anexa A) și în fișierul **IO.ASM** (fișier sursă, listat în Anexa B). Aceste fișiere asigură (între altele) suport pentru următoarele operații de bază:

- introducerea de caractere de la consolă;
- afișarea de caractere la consolă;
- introducerea de șiruri de caractere de la consolă;
- afișarea de șiruri de caractere la consolă;
- afișarea unor mesaje imediate la consolă;
- introducerea de întregi pe 16 biți cu sau fără semn de la consolă;
- afișarea de întregi pe 16 biți cu sau fără semn la consolă;
- inițializarea registrelor DS și ES la intrarea în program;
- terminarea programului cu ieșire în sistemul DOS.

Se vor utiliza directivele de definire simplificată a segmentelor. Șablonul de dezvoltare al unui program ASM care conține un modul executabil va fi următorul:

```
.model      MMMMM
include     io.h
```

```
.stack      NNNN
.data
            ; Definitii de date
.code
            ; Definitii de proceduri
start:
      init_ds_es
            ; Program principal
      exit_dos
end   start
```

unde:

- **MMMMM** este modelul de memorie, care poate fi **tiny**, **small**, **medium**, **compact**, **large** sau **huge**; modelele uzuale sunt **small** și **large**, în care toate adresele, procedurile, apelurile, salturile și revenirile din procedură sunt implicit de tip NEAR, respectiv de tip FAR;
- **NNNN** este dimensiunea rezervată segmentului de stivă; o valoare uzuală este 1024;
- **include io.h** este o directivă care include în textul sursă fișierul io.h, care trebuie să se găsească în același catalog (director) cu fișierul sursă;
- **init_ds_es** este o macroinstrucțiune (definită în io.h) care inițializează registrele DS și ES cu adresa segmentului de date; registrele SS și CS sunt inițializate automat la încărcarea programului executabil de pe disc.
- **exit_dos** este o macroinstrucțiune (definită în io.h) care provoacă terminarea programului și revenirea în sistemul DOS;
- **start** este o etichetă care marchează punctul de început al programului principal;
- **end start** este o directivă care precizează că modulul curent este modul de program principal, având punctul de intrare la eticheta start.

O altă variantă este scrierea programului principal sub forma unei proceduri (având numele, de exemplu, _main) și precizarea punctului de start prin numele procedurii:

```
.model      MMMMM
include     io.h
.stack      NNNN
.data
            ; Definitii de date
.code
            ; Definitii de proceduri
_main proc
      init_ds_es
            ; Program principal
      exit_dos
_main endp
end   _main
```

Un modul de program care nu este modul de program principal (deci conține definiții de date și/sau proceduri) nu are etichetă în directiva end. Într-o aplicație dezvoltată modular (în mai multe fișiere sursă), un singur modul poate fi modul de program principal.

Asamblorul nu face distincție între simboluri scrise cu litere mici sau mari. De acum încolo, vom scrie de regulă programele cu litere mici, iar cu litere mari unele directive și tipuri de date definite de utilizator, pentru a le scoate în evidență.

Asamblarea unui fișier sursă NUME.ASM se face cu una din comenzile:

```
C:\> tasm nume.asm
C:\> tasm nume
```

în urma căreia rezultă un fișier obiect NUME.OBJ.

Dacă se dorește și fișier listing, se dă comanda:

```
C:\> tasm nume ,,nume
```

în urma căreia rezultă și un fișier NUME.LST.

Dacă se dorește o depanare simbolică ulterioară, trebuie introdusă opțiunea /zi:

```
C:\> tasm nume /zi
```

Operația de asamblare se face pentru fiecare fișier sursă în parte.

Legarea modulelor se face cu comanda:

```
C:\> tlink nume_1 nume_2 ... [, nume_bin ] [/v]
```

în care parantezele drepte indică parametrii opționali; nume_1, nume_2 sunt numele modulelor obiect, nume_bin este numele fișierului executabil rezultat (dacă lipsește, se consideră numele primului modul obiect), iar /v este o opțiune de depanare simbolică (se introduce dacă vrem să executăm programul sub controlul depanatorului TD). În urma comenzii, rezultă un fișier executabil.

O situație tipică este cea a unui singur modul sursă, caz în care comanda de legare va fi:

```
C:\> tlink nume io
```

prin care se leagă și modulul io.obj (care conține procedurile de intrare-ieșire descrise mai sus).

Peste tot în cuprinsul cărții vom considera că șirurile de caractere sunt terminate cu octetul 0 binar. Definiția unui șir constant sau rezervarea de spațiu pentru un șir variabil se pot face prin directiva Define Byte (constantele simbolice cr și lf sunt definite în fișierul io.h). Definiția unui întreg sau rezervarea de spațiu pentru

un întreg se poate face cu directiva Define Word. Definirea de spațiu la nivel de caracter se face cu directiva Define Byte.

```
.data
        sir_1       db      'Un sir de caractere', cr, lf, 0
        sir_2       db      80 dup (0)
        i_1         dw      -200
        i_2         dw      ?
        u_1         dw      0FFFFH
        u_2         dw      -1
        c_1         db      'A'
        c_2         db      ?
```

Afișarea unui șir de caractere la consolă se poate face prin macroinstrucțiunea **puts (Put String)**, în una din formele:

```
puts    sir_1           ; O prima forma de invocare
lea     si, sir_1       ; O a doua
puts    [si]            ; forma de invocare
```

Citirea unui șir de caractere de la consolă se poate face cu macroinstrucțiunea **gets (Get String)**, în una din formele:

```
gets    sir_2           ; O prima forma de invocare
lea     bx, sir_2       ; O a doua
gets    [bx]            ; forma de invocare
```

Afișarea unui șir constant de caractere (mesaj la consolă) se poate face cu macroinstrucțiunea **putsi (Put String Immediate)**, care nu necesită definirea șirului și nici prezența explicită a terminatorului 0:

```
putsi           <'Introduceti un sir de caractere', cr, lf>
gets    sir_2
putsi           <'Sirul introdus este:', cr, lf>
puts    sir_2
putsi           <cr,lf>
```

Introducerea unui întreg cu sau fără semn se poate face cu macroinstrucțiunile **geti (Get Integer)**, fără parametri, care întoarce întregul citit în registrul AX.

Afișarea unui întreg cu sau fără semn se poate face cu macroinstrucțiunile **puti (Put Integer)** sau **putu (Put Unsigned)**, în una din formele:

```
geti                    ; Citeste intreg in AX
puti    ax              ; Afiseaza intregul din AX
mov     i_1, ax         ; Depune
puti    i_1             ; Afiseaza intreg cu semn din memorie
putu    u_1             ; Afiseaza ca numar fara semn
lea     di, i_2
puti    [di]            ; Afiseaza ca numar cu semn
putu    [di]            ; Afiseaza ca numar fara semn
```

Introducerea unui singur caracter de la consolă se poate face cu macroinstrucțiunea **getc (Get Character)**, care întoarce caracterul în registrul AL, iar afișarea unui caracter se poate face cu macroinstrucțiunea **putc (Put Character)**, în una din formele:

```
putc  'A'
putc  car_1
lea   bx, car_2
putc  [bx]
```

Toate macroinstrucțiunile de mai sus conservă registrele procesorului, deci nu e nevoie de salvări și restaurări explicite.

Să considerăm un exemplu de program, în care se fac următoarele operații:

- se citesc de la consolă cel mult 20 de întregi cu semn;
- se afișează valorile introduse;
- se sortează crescător aceste valori;
- se afișează valorile sortate.

Considerăm modelul de memorie **large**, adică toate adresele sunt implicit de 32 de biți. Pentru sortare, vom folosi un algoritm elementar (metoda bulelor), descris mai jos (se consideră că indicii din tabloul **a** variază de la 0 la n-1):

```
for i = 1 to n-1
for j = n - 1 downto i
      if ( a[j-1] > a[j] )
            schimba a[j] cu a[j-1]
```

Implementarea întregului program este următoarea:

```
.model      large
include     io.h
.stack      1024
.data
      vec   dw    20 dup (?)
      n     dw    ?
.code
tipvec proc far
;     Procedura de afisare vector.
;     Date de intrare:
;     ds:si = adresa vector de intregi
;     cx = numar de elemente

      jcxz  tipend            ; Nu avem ce afisa
tip:
      puti  [si]              ; Afisare intreg cu semn
      putsi <' '>             ; Spatiu
      add   si, 2             ; Actualizare adresa
      loop  tip               ; Bucla dupa CX
tipend:
       ret
tipvec endp
```

```
bubble proc far
;       Procedura de sortare
;       Date de intrare
;       ds:bx = adresa tablou de intregi
;       cx = dimensiune tablou
;       Variabila i : asignata la si
;       Variabila j : asignata la di
;       Adresa de inceput a tabloului: asignata la bx

        cmp     cx, 1
        jbe     algend                  ; Nu avem ce sorta
        mov     si, 1                   ; i = 1
fori:
        mov     di, cx
        dec     di                      ; j = n-1
forj:
        shl     di, 1                   ; intregii pe 2 octeti
        mov     ax, [bx][di-2]          ; a[j-1]
        cmp     ax, [bx][di]            ; Compara cu a[j]
        jle     nextj                   ; Mai mic sau egal
        xchg    ax, [bx][di]            ; Schimba a[j]
        mov     [bx][di-2], ax          ; cu a[j-1]
nextj:
        shr     di, 1                   ; Refacere indice
        dec     di                      ; Ciclu downto
        cmp     di, si
        jae     forj                    ; Cat timp j >= i
nexti:
        inc     si                      ; Ciclu to
        cmp     si, cx
        jb      fori                    ; Cat timp i < n
algend:
        ret
bubble endp

;       Programul principal

start:
        init_ds_es
        putsi <'Introduceti datele',cr,lf>
        mov     cx, 20              ; Numar maxim
                                    ; de elemente
        lea     bx, vec             ; Adresa tablou
iar:
        geti                        ; Citire intreg cu semn
        test    ax, ax              ; Este 0 ?
        jz      gata                ; Daca da, gata
        mov     [bx], ax            ; Depunere in tablou
        add     bx, 2               ; Actualizare adresa
        loop    iar                 ; Bucla dupa CX
gata:
        mov     ax, 20              ; Calcul numar de
        sub     ax, cx              ; elemente introduse
        mov     n, ax
```

```
        putsi       <'Vector nesortat', cr, lf>
        lea   si, vec           ; Adresa tablou
        mov   cx, n             ; Numar de elemente
        call  tipvec            ; Afisare tablou
                                ; nesortat

        lea   bx, vec           ; Adresa tablou
        mov   cx, n             ; Numar de elemente
        call  bubble            ; Sortare tablou

        putsi <cr,lf,'Vector sortat', cr, lf>
        lea   si, vec
        mov   cx, n
        call  tipvec            ; Afisare tablou sortat
        exit_dos                ; Iesire in DOS
end     start
```

În segmentul de date, se rezervă spațiu pentru tabloul vec de cel mult 20 de întregi și pentru dimensiunea n a acestuia.

Procedura TIPVEC primește în SI adresa unui tablou și în CX numărul de elemente, realizând afișarea la consolă a elementelor, considerate întregi cu semn. Se observă forma standard a unei bucle de program implementată cu instrucțiunea LOOP.

Procedura BUBBLE implementează algoritmul de sortare. Indicii i și j din descrierea algoritmului sunt asignați la regiștrii SI și DI. Indicii sunt incrementați sau decrementați cu 1, dar întregii sunt pe doi octeți. De aceea, în secvența de adresare a memoriei, se înmulțește temporar DI cu 2, pentru a accesa corect elementele tabloului. Astfel, elementul de indice 0 se va afla la deplasament 0, elementul de indice 1 la deplasament 2 etc. Înmulțirea și refacerea se realizează prin deplasări logice. Se observă secvența standard de interschimbare a două elemente din memorie (două MOV-uri și un XCHG). Trebuie remarcat modul în care se fac comparațiile diverselor elemente: indicii sunt considerați fără semn, deci testele se fac cu instrucțiuni de salt condiționat de tip „Above" sau „Below". În schimb, tabloul este cu semn, deci comparațiile între elemente se fac cu instrucțiuni de salt condiționat de tip „Greater" sau „Less". Programul principal începe printr-o buclă de citire a datelor. Introducerea se poate opri mai devreme de 20 de elemente dacă se introduce valoarea 0. La ieșirea din buclă, se calculează numărul de elemente efectiv introduse, ca diferența dintre numărul maxim admis (20) și valoarea curentă a lui CX. Acest număr se depune în variabila n, pentru utilizări viitoare. În continuare, vom avea secvențe de apel ale procedurilor TIPVEC și BUBBLE. Se observă încărcarea corespunzătoare a parametrilor (adresa tabloului și numărul de elemente) în registrele desemnate în acest scop pentru fiecare procedură. Exemplul prezentat ilustrează modul de utilizare a macroinstrucțiunilor prezentate, ca și schema generală de dezvoltare a unui program.

Capitolul 3

Procesoare de 32 de biți
Coprocesoare matematice

3.1 Arhitectura procesoarelor de 32 de biți

3.1.1 Generalități

Apariția și dezvoltarea procesoarelor Intel de 32 de biți a constituit o adevărată revoluție în domeniul calculatoarelor personale, atât prin creșterea performanțelor hardware (în special creșterea spațiului de memorie adresabilă direct), cât și prin perspectivele deschise sistemelor de programe.

Procesoarele de 32 de biți sunt compatibile ca arhitectură cu cele de 16 biți, prin aceea că registrele de 16 biți se regăsesc ca subregistre ale registrelor de 32 de biți. Aceasta a permis ca setul de instrucțiuni al procesoarelor de 16 biți să fie un subset al celui specific procesoarelor de 32 biți, ceea ce face ca orice program dezvoltat pentru procesoarele de 16 biți să poată fi executate și pe mașini de 32 de biți. În plus, toate instrucțiunile care utilizau operanzi de 16 biți se extind și asupra operanzilor de 32 de biți. Această compatibilitate a setului de instrucțiuni a fost unul din scopurile declarate ale noii generații de procesoare, pentru a conserva imensa cantitate de software dezvoltată anterior.

Figura 3.1 descrie registrele procesoarelor de 32 de biți, din punctul de vedere al programelor de aplicație.

Există opt registre generale de 32 de biți, care au numele registrelor generale de 16 biți, cu prefixul E (de la extended). Astfel, registrul EAX se numește acumulator extins, ESP se numește stack pointer extins etc. Primii 16 biți mai puțin semnificativi ai acestor registre sunt disponibili și ca registre de 16 biți (AX, SP etc.), fiind practic identice cu registrele generale 8086.

Aceeași abordare se regăsește și în cazul registrelor EIP (contor program extins) și EFLAGS (registru de flaguri extins). Subsetul de bistabili din registrul FLAGS se regăsește în partea low a registrului EFLAGS.

Registrele de segment au fost păstrate de 16 biți, dar s-au adăugat două noi registre: FS și GS.

Pe lângă registrele generale din Figura 3.1, procesoarele de 32 de biți dispun de registre de control, de gestiune a adresei, de depanare și de test. Acestea diferă

de la un tip de procesor la altul și sunt folosite în principal de programele de sistem.

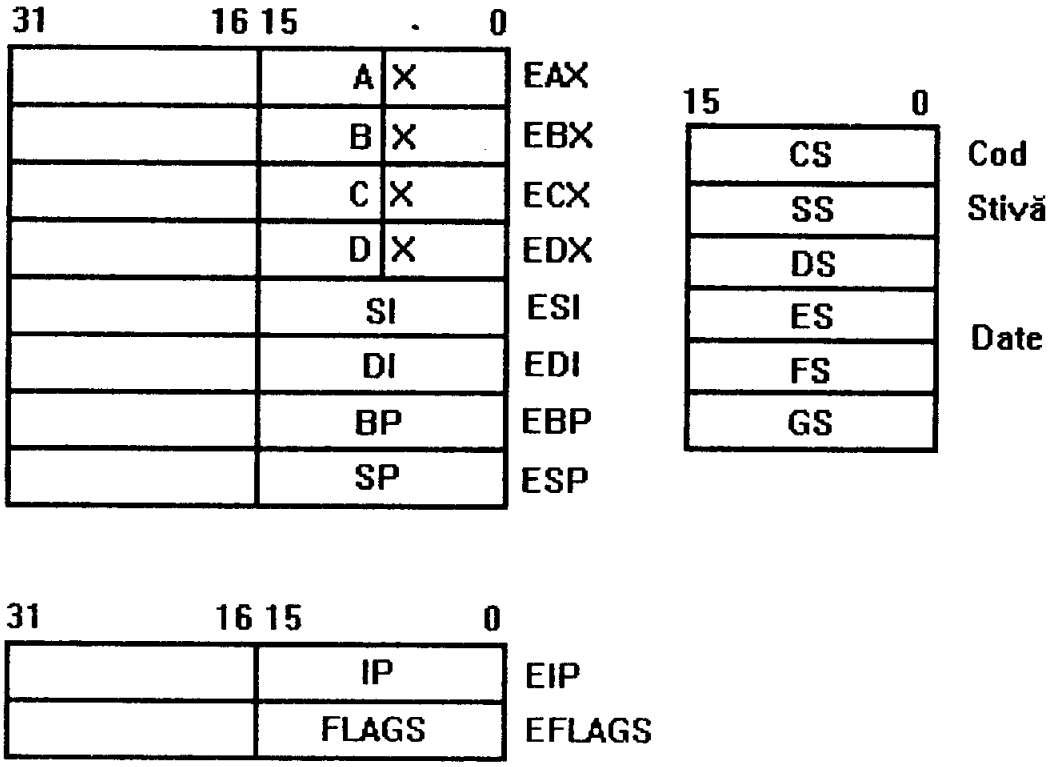


Figura 3.1 Registrele procesoarelor de 32 de biți

Una din modificările majore ale arhitecturii este semnificația registrelor de segment, care acționează în general ca selectoare de segment. Acestea nu mai indică în mod nemijlocit adresa de bază a segmentului de memorie, ci un descriptor de segment (încărcat într-un registru special de control) care precizează adresa de bază a segmentului, dimensiunea acestuia și drepturile de acces asociate. Abordarea respectivă permite ca adresa de bază și limita segmentului să poată fi specificate pe 32 de biți, crescând dimensiunea maximă a unui segment până la 4 GB.

3.1.2 Moduri de adresare pe 32 de biți

Modurile de adresare (formarea adresei fizice) au fost mult dezvoltate față de cele pe 16 biți. Se introduc următoarele noțiuni:

- **deplasament** - o valoare imediată pe 8 sau 32 de biți, conținută în instrucțiune;
- **registru de bază** - orice registru general de 32 de biți (spre deosebire de adresarea pe 16 biți unde ca registre de bază se utilizează doar BX și BP);

- **registru index** - orice registru general de 32 de biți, cu excepția lui ESP (spre deosebire de adresarea pe 16 biți unde ca registre index se utilizează doar SI și DI);
- **factor de scală** - indexul poate fi înmulțit cu un factor de scală de valoare 1, 2, 4 sau 8 (inexistent în adresarea pe 16 biți).

Se obțin astfel 9 moduri posibile de adresare:

- **adresare directă** - adresa efectivă a operandului face parte din instrucțiune, putând fi pe 8, 16 sau 32 de biți; exemplu:

```
INC dword ptr [1000H]
```

- **adresare indirectă prin registre** - adresa efectivă a operandului este conținută într-unul din registrele de bază; exemplu:

```
MOV [EBX], EAX
```

- **adresare bazată** - adresa efectivă a operandului este formată din conținutul unui registru de bază la care se poate adăuga un deplasament; exemplu:

```
ADD ECX, [EAX+32]
```

- **adresare indexată** - adresa efectivă a operandului este formată din conținutul unui registru index la care se poate adăuga un deplasament; exemplu:

```
MUL byte ptr TABLOU [ESI]
```

- **adresare indexată cu factor de scală** - adresa efectivă a operandului este formată din conținutul unui registru index, înmulțit cu un factor de scală, la care se poate adăuga un deplasament; exemplu:

```
MOV EAX, dword ptr TABLOU [EDI*4][100H]
```

- **adresare bazată și indexată** - adresa efectivă a operandului este formată din conținutul unui registru de bază la care se adună conținutul unui registru index; exemplu:

```
MOV EAX, [ESI][EBX]
```

- **adresare bazată și indexată cu factor de scală** - adresa efectivă a operandului este formată din conținutul unui registru de bază la care se adaugă conținutul unui registru index, înmulțit cu un factor de scală; exemplu:

```
MOV ECX, [EDX*8][EAX]
```

- **adresare bazată și indexată cu deplasament** - adresa efectivă a operandului este formată din conținutul unui registru de bază la care se adaugă conținutul unui registru index, la care se poate adăuga un deplasament; exemplu:

```
ADD EDX, [ESI] [EBP + 10000H]
```

- **adresare bazată și indexată, cu factor de scală și deplasament** - adresa efectivă a operandului este formată din conținutul unui registru de bază la care se adaugă conținutul unui registru index, înmulțit cu un factor de scală, la care se poate adăuga un deplasament; exemplu:

```
MOV EAX, TABLOU [EDI*4] [EBP+800H]
```

Figura 3.2 ilustrează modul cel mai general de adresare (bazată și indexată, cu factor de scală și deplasament), în care adresa efectivă AE se calculează după relația:

```
AE = Registru_Baza + Registru_Index * Factor_Scala + Deplasament
```

Factorul de scală este foarte util la parcurgerea tablourilor de date simple (pe 2, 4 sau 8 octeți), permițând ca indicele logic al tabloului să coincidă cu conținutul registrului index. De exemplu, o instrucțiune C de forma:

```
long int tablou [1000];
for (i = 0; i < 1000; i++)
      tablou[i] += 10;
```

se implementează prin secvența:

```
      MOV   CX, 1000
      XOR   EBX, EBX
bucla:
      ADD   _TABLOU [EBX*4], 10
      INC   EBX
      LOOP  bucla
```

în care registrul EBX are exact aceeași semnificație ca și indicele i din programul C.

3.1.3 Modurile Real și Protected

Pentru a păstra compatibilitatea cu programele dezvoltate pentru mașini de 16 biți, procesoarele de 32 de biți dispun de două moduri de operare: modul **Real Address** (pe scurt, modul **Real**) și modul **Protected Virtual Address** (pe scurt, modul **Protected**).

În modul **Real**, procesorul se comportă ca un procesor de 16 biți foarte rapid, permițând însă accesul la date și adrese de 32 de biți.

Modul **Protected** oferă mecanisme sofisticate de gestiune a memoriei (paginare, drepturi de acces la segmente etc.). În acest mod de operare, unitățile de program se numesc taskuri. Se poate executa o comutare de task, pentru a intra într-un regim special de funcționare, denumit mod Virtual 8086. Fiecare asemenea task se comportă (din punct de vedere software) ca o mașină

de tip 8086, permițând programelor de 16 biți (aplicații sau chiar un întreg sistem de operare) să poată fi executate ca taskuri.

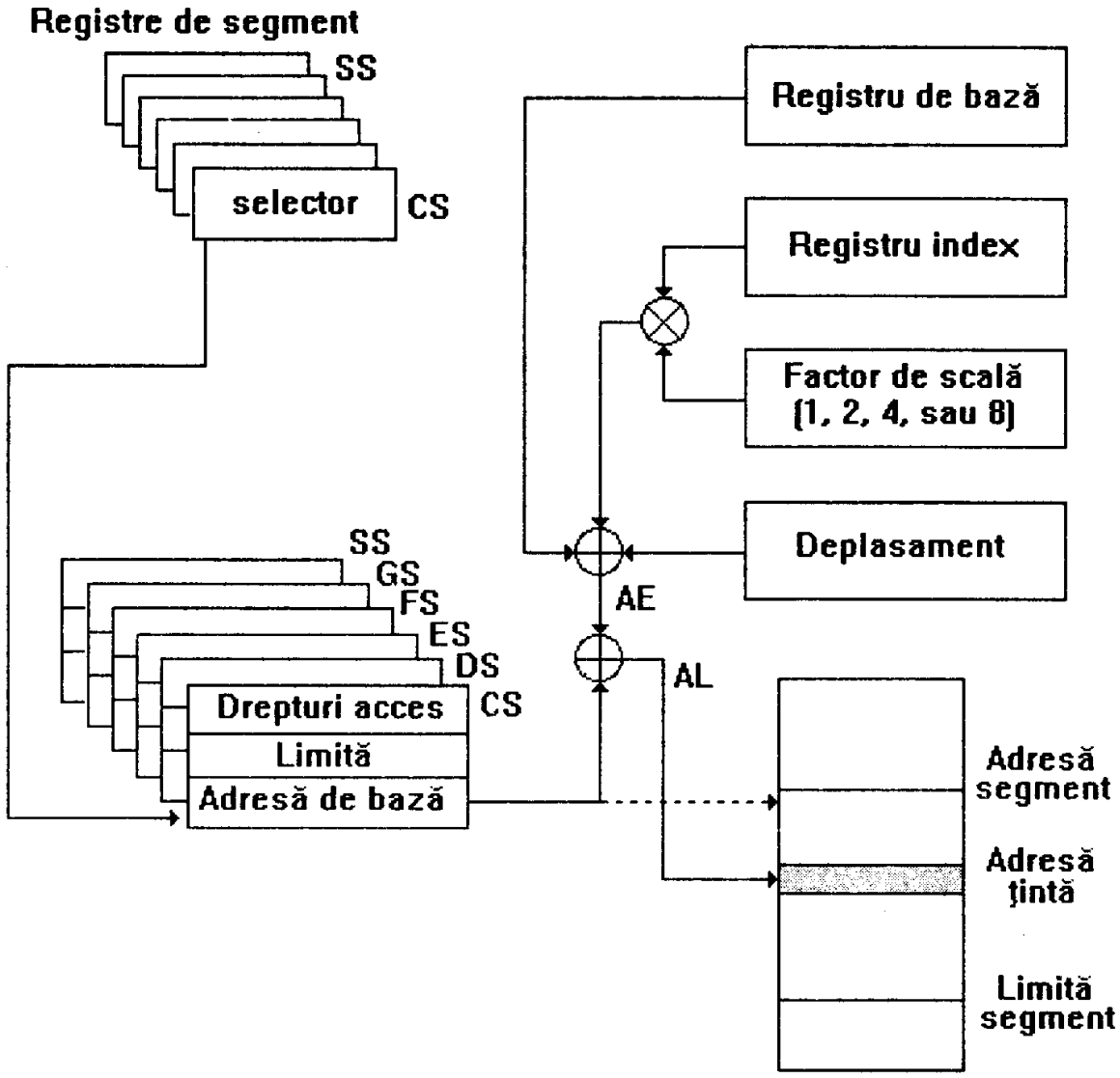


Figura 3.2 Moduri de adresare pe 32 de biți

Atât în modul **Real**, cât și în modul **Protected**, procesorul poate executa instrucțiuni de 16 biți, prin examinarea unui bit din descriptorul de segment asociat registrului CS. Acest bit precizează lungimea implicită a operanzilor și a adreselor (16 sau 32 de biți).

În afara dimensiunii implicite, se poate forța o anumită dimensiune prin prefixele numite **Operand Size** (Dimensiune operand) și **Address Length** (Lungime adresă). Aceste prefixe (de valoare 66H și 67H) sunt generate automat de către macroasambloare. Să presupunem modul Real și să considerăm următoarea secvență de cod sursă:

```
.code
        MOV     AX, word ptr [DI][100H]
        MOV     EAX, dword ptr [DI][100H]
```

```
MOV    AX, word ptr [EDI][100H]
MOV    EAX, dword ptr [EDI][100H]
```

Fișierul listing generat va conține următoarea descriere:

```
0000 8B   85   0100             MOV AX, word ptr [DI][100H]
0004 66|  8B   85 0100          MOV EAX, dword ptr [DI][100H]
0009 67|  8B   87 00000100      MOV AX, word ptr [EDI][100H]
0010 66|  67|  8B 87 00000100 MOV EAX, dword ptr [EDI][100H]
```

Se observă același cod al instrucțiunii MOV (8BH), același deplasament 100H generat pe 16 sau 32 de biți și prefixele inserate automat. Prima instrucțiune este cu operand și adresare de 16 biți. A doua instrucțiune este cu operand pe 32 de biți și adresare pe 16 biți; se observă prefixul 66H și offset-ul 100H generat pe 16 biți. A treia instrucțiune este cu operand pe 16 biți și adresare pe 32 de biți; se observă prefixul 67H și offset-ul 100H generat pe 32 de biți. În fine, ultima instrucțiune este cu operand și adresare pe 32 de biți; se observă ambele prefixe și offset-ul 100H generat pe 32 de biți.

În modul de operare **Real**, formarea adresei fizice este exact ca la procesoarele de 16 biți (vezi Figura 3.3). Selectorul de segment este unul din cele 4 registre de segment, iar offset-urile sunt pe 16 biți, conducând astfel la o lungime maximă a unui segment de 64KB. Pointerii către date sau către instrucțiuni pot fi memorați pe 4 octeți (doi octeți pentru adresa de segment și doi octeți pentru offset).

Asambloarele dispun de directiva DD (Define DoubleWord), care generează pointeri pe 4 octeți, ca în exemplul următor:

```
.data
     TABLOU       DD     256 dup (?)
     ADR_16       DD     TABLOU
.code
     LES    BX, ADR_16
     MOV    EAX, ES:[BX]
```

Instrucțiunea LES BX, ADDR_16 încarcă perechea de registre (ES:BX) cu un pointer pe 4 octeți.

În modul de operare **Protected**, registrele selectoare de segment (de 16 biți) sunt încărcate cu adresa unui descriptor de segment, iar offset-urile pot fi de 16 sau 32 de biți (vezi Figura 3.4), conducând la o lungime maximă a unui segment de 4 GB. Pointerii pot fi pe 4 octeți (doi octeți pentru descriptorul de segment și doi octeți pentru offset) sau pe 6 octeți (patru octeți pentru offset).

Asambloarele dispun de directiva DP (Define Pointer), care generează pointeri pe 6 octeți, ca în exemplul următor:

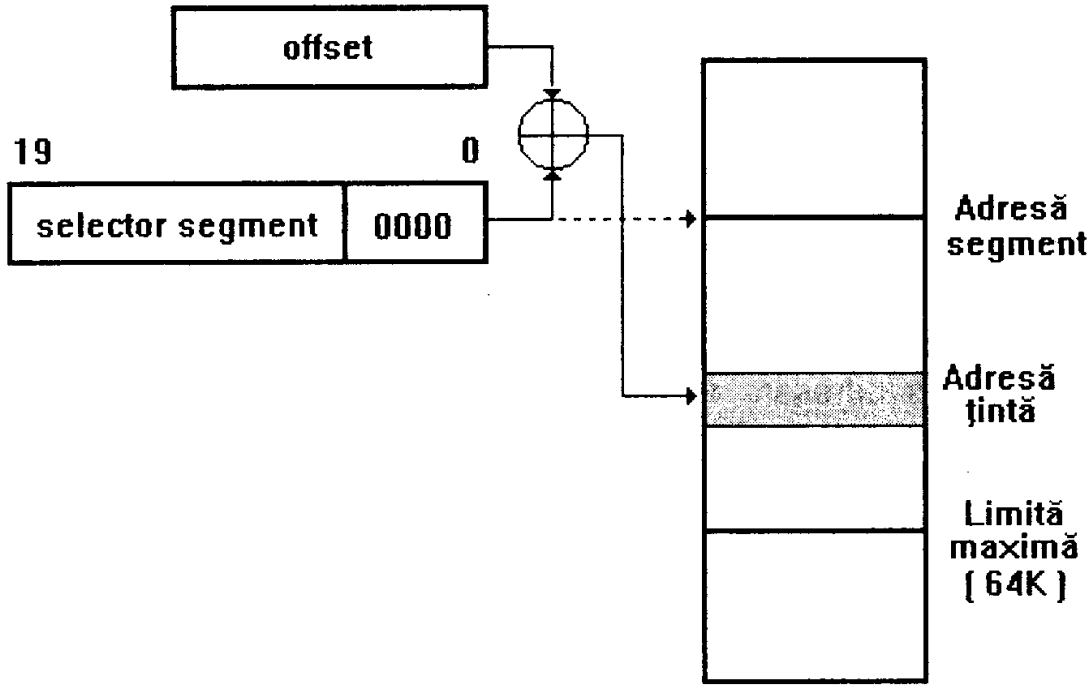


Figura 3.3 Adresarea în modul Real

```
.data
      TABLOU      DD    256 dup (?)
      ADR_32      DP    TABLOU
.code
      LFS         EBX, ADR_32
      MOV         EAX, FS:[EBX]
```

Instrucțiunea LFS EBX, ADDR_32 încarcă în perechea de registre (FS:EBX) cu un pointer pe 6 octeți.

Să considerăm următoarea secvență sursă:

```
.data
      TABLOU      DD    256 dup (?)
      ADR_16      DD    TABLOU
      ADR_32      DP    TABLOU
.code
      LEA   BX, TABLOU        ; Acces prin pointer near
                              ; de 16 biti in
      MOV   EAX, [BX]         ; cadrul  segmentului selectat
                              ; prin DS
      LEA   EBX, TABLOU       ; Acces prin pointer near
                              ; de 32 de biti in
      MOV   EAX, [EBX]        ; cadrul segmentului selectat
                              ; prin DS
      LES   AX, ADR_16        ; Acces prin pointer far
                              ; de 32 de biti în
      MOV   EAX, ES:[BX]      ; cadrul segmentului selectat
                              ; prin ES
```

```
LFS    EBX, ADR_32          ; Acces prin pointer far
                            ; de 48 de biti in
MOV    EAX, FS:[EBX]        ; cadrul segmentului selectat
                            ; prin FS
```

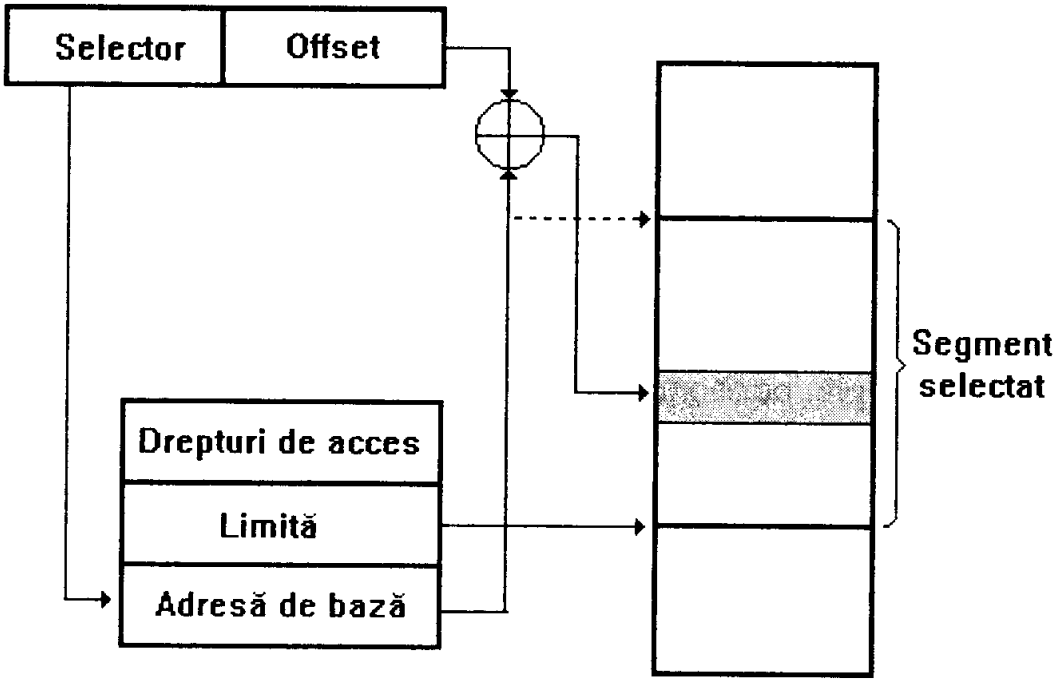


Figura 3.4 Adresarea în modul Protected

Listingul generat la asamblare pentru secvența de mai sus este:

```
0000                    .data
0000 0100*(????????)            TABLOU      DD 256 dup (?)
0400 00000000sr                 ADR_16      DD TABLOU
0404 000000000000sr             ADR_32      DP TABLOU

040A                    .code
0000 BB 0000r                   LEA    BX, TABLOU
0003 66| 8B 07                  MOV    EAX, [BX]

0006 66| 8D 1E 0000r            LEA    EBX, TABLOU
000B 66| 67| 8B 03              MOV    EAX, [EBX]

000F C4 1E 0400r                LES    BX, ADR_16
0013 66| 26: 8B 07              MOV    EAX, ES:[BX]

0017 66| 0F B4 1E 0404r         LFS    EBX, ADR_32
001D 66| 64: 67| 8B 03          MOV    EAX, FS:[EBX]
```

Se observă prezența a trei categorii de prefixe. Prefixele de operand pe 32 de biți (marcate cu 66|), cele de adresă pe 32 de biți (marcate cu 67|) și prefixele de segment (marcate cu 26: pentru ES: și cu 64: pentru FS:). Instrucțiunea LEA

(Load Effective Address) permite încărcarea atât a offset-urilor de 16 biți, cât și a celor de 32 de biți.

În mod implicit, accesul la date se face prin selectorul DS (la fel ca la 8086), cu excepția modurilor de adresare care implică registrele EBP și ESP, caz în care selectorul implicit este SS. Se poate folosi orice prefix de segment, cu aceleași excepții cunoscute de la 8086:

- instrucțiunile executate sunt accesate totdeauna prin CS;
- operațiile de salvare și restaurare în stivă implicate în instrucțiunile CALL, RET, PUSH și POP se execută totdeauna prin selectorul SS;
- destinația instrucțiunilor pentru șiruri este accesată totdeauna prin selectorul ES.

La punerea sub tensiune, procesorul este în modul **Real**. Trecerea în modul **Protected** se face prin poziționarea unui bit din cuvântul mai puțin semnificativ (numit **Machine Status Word**) al registrului de control CR0.

Așa cum la procesoarele de 16 biți trebuia să încărcăm registrele de segment cu valori cu sens, trecerea în modul **Protected** presupune definirea corespunzătoare a descriptorilor de segment și încărcarea registrelor selectoare. Aceste operații nu vor fi descrise, în ideea că programele de aplicație dezvoltate în limbaj de asamblare vor opera în general în modul **Real**.

3.2 Setul de instrucțiuni al procesoarelor de 32 de biți

3.2.1 Extensia instrucțiunilor de 16 biți

Pentru ca asamblorul să recunoască instrucțiunile specifice, se utilizează directivele .286, .386. .486, care precizează tipul procesorului.

O serie de restricții de la setul de bază 8086 sunt eliminate, începând de la procesorul 80286. De asemenea, se adaugă o serie de instrucțiuni noi.

Încărcarea registrelor de segment se poate face și cu valori imediate; exemplu MOV DS, DGROUP.

Instrucțiunile de deplasare și rotație se pot executa pe un număr oarecare de biți, specificat în instrucțiune; de exemplu: SHL AX, 4.

Instrucțiunea de înmulțire cu semn (IMUL) are următoarele forme suplimentare:

- **IMUL reg16, im16**

prin care se înmulțește registrul de 16 biți reg16 cu o valoare imediată pe 16 biți im16, iar rezultatul se depune în reg16;

- **IMUL dreg16, sursa16**

prin care se înmulțește dreg16 cu sursa16 (registru sau memorie) și rezultatul se depune în dreg16 (numai la 80386 și următoarele);

- **IMUL dreg16, sursa16, im16**

prin care se înmulțește sursa16 (registru sau memorie) cu valoarea imediată im16 și rezultatul se depune în registrul dreg16 (destinație).

Instrucțiunile de lucru cu stiva se extind cu:

- **PUSHA** (Push All)

care salvează în stivă registrele AX, CX, DX, BX, SP, BP, SI, DI (în această ordine), iar registrul SP este scăzut cu 16;

- **POPA** (Pop All)

care reface registrele DI, SI, BP, SP, BX, DX, CX, AX (salvate cu o instrucțiune PUSHA anterioară); registrul SP se reface prin adunarea valorii 16, nu prin extragerea efectivă din stivă;

- **PUSH** im16

care pune în stivă o valoare imediată.

Instrucțiunile pentru șiruri se extind cu:

- **INSB, INSW** (Input String)

cu semnificația:

(ES:(DI)) ← byte/word citit de la portul specificat de DX;

actualizează DI;

- **OUTSB, OUTSW** (Output String)

cu semnificația:

port specificat prin DX ← (DS:SI));

actualizează SI;

Instrucțiunile de control se extind cu:

- **BOUND reg16, lim** (Testează limite)

în care lim este adresa unei zone de memorie de doi întregi, care precizează limita minimă și maximă pentru registrul reg16. Dacă reg16 < DS:(lim) sau reg16 > DS:(lim+2), atunci se generează o întrerupere software pe nivelul 5.

- **ENTER dim_loc, nivel** (Creează șablon stivă la intrarea în procedură)

în care dim_loc și nivel sunt întregi pe 16 biți fără semn.

Semnificația este următoarea:

```
PUSH BP
temp ← SP
daca (nivel > 0) {
      for (i = 1 to nivel-1) {
            BP ← BP - 2
            PUSH SS:[BP]
      }
      PUSH temp
}
```

```
BP ← temp
SP ← SP - dim_loc
```

Primul parametru dim_loc este dimensiunea necesară în stivă pentru parametrii locali ai procedurii. Dacă nivel este 0, atunci semnificația este echivalentă cu:

```
PUSH  BP
MOV   BP, SP
SUB   SP, dim_loc
```

care este secvența standard de intrare într-o procedură cu parametri locali în stivă (vezi 6.6).

Dacă nivel este > 0, se copiază în stivă conținutul zonei de stivă (parametri locali) a procedurii apelante. Dacă procedura curentă este apelată dintr-o altă procedură, care are registrul BP poziționat printr-o instrucțiune ENTER, atunci șablonul stivei curente va conține primii nivel-1 parametri locali ai procedurii apelante.

- **LEAVE** (Pregătire pentru ieșire din procedură)

Instrucțiunea LEAVE reface registrele SP și BP, anulând efectul instrucțiunii LEAVE. Este echivalentă cu secvența:

```
MOV SP, BP
POP BP
```

Instrucțiunea ENTER se scrie ca primă instrucțiune din procedură, iar LEAVE imediat înainte de RET.

3.2.2 Instrucțiuni specifice procesoarelor de 32 de biți

Întregul set de instrucțiuni de transfer, aritmetice și logice se extinde cu operanzi de tip DoubleWord (registre, memorie sau valori imediate).

Instrucțiuni specifice de transfer

- **PUSHAD/POPAD** (Push/Pop All Double)

Salvează/restaurează registrele EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI în stivă.

- **PUSHFD/POPFD** (Push Flags Double)

Salvează/restaurează registrul EFLAGS în stivă.

- **CMPXCHG reg/mem, reg** (Compare and Exchange)

Compară și apoi interschimbă cei doi operanzi, poziționând flagurile aritmetice.

- **BSWAP reg32 (Byte Swap)**

Inversează ordinea octeților într-un registru de 32 de biți.

- **XADD reg/mem, reg (Exchange and Add)**

Este echivalentă cu secvența:

```
XCHG  reg/mem, reg
```

```
ADD   reg/mem, reg
```

Instrucțiuni specifice de conversie

- **MOVZX/MOVSX** destinație, sursă (Move with Zero/Sign Extension)

Copiază operandul sursă (de 8 sau 16 biți) în operandul destinație (de lungime dublă), completând cu 0 (MOVZX) sau cu bitul de semn al operandului sursă (MOVSX). Destinația poate fi registru sau memorie de 16/32 de biți, iar sursa poate fi registru, memorie sau valoare imediată de 8/16 biți.

- **CWDE** (Conver Word to DoubleWord Extended)

Convertește AX la un dublu cuvânt în EAX, cu extensie de semn. Este asemănătoare cu CWD (Convert Word to Double), care are ca destinație perechea (DX:AX).

- **CDQ** (Convert DoubleWord to QuadWord)

Convertește registrul EAX la un cuvânt cvadruplu în perechea (EDX:EAX), cu extensie de semn.

Instrucțiuni specifice adreselor

- **LFS/LGS/LSS** registru, memorie

Încarcă o pereche de registre cu un pointer din memorie, pe 4 sau 6 octeți. Registrul specificat poate fi de 16 sau de 32 de biți.

Instrucțiuni aritmetice

Operațiile de înmulțire și de împărțire se pot executa pe 32 de biți. Se utilizează registrele EAX și EDX, cu semnificație asemănătoare ca la operațiile de 16 biți. De exemplu, instrucțiunea:

```
DIV   EBX
```

împarte (EDX:EAX) la EBX, cu câtul în EAX și restul în EDX.

Instrucțiuni de deplasare

- **SHLD/SHRD** reg/mem, reg, CL/im8 (Shift Left/Right Double)

Se deplasează reg/mem la stânga/dreapta cu numărul de biți specificat în CL sau în operandul imediat im8 (pe 8 biți). Al doilea operand este folosit pentru a completa biții deplasați ai primului operand. Primii doi operanzi trebuie să fie de aceeași dimensiune (Word sau DoubleWord).

Instrucțiuni pentru șiruri

Setul de instrucțiuni pentru șiruri se extinde cu operații specifice șirurilor de întregi pe 4 octeți. Există astfel instrucțiunile MOVSD (Move String Double), STOSD (Store String Double), LODSD (Load String Double), SCASD (Scan String Double). Se utilizează registrul acumulator extins EAX, iar adresele sursă și destinație (SI și DI) sunt incrementate/decrementate cu 4.

Instrucțiuni la nivel de bit

- **BSF/BSR reg, reg/mem** (Bit Scan Forward/Reverse)

Instrucțiunea parcurge operandul sursă reg/mem (16 sau 32 de biți), până la primul bit egal cu 1. BSF parcurge de la dreapta la stânga, iar BSR invers. Indicele bitului 1 identificat este plasat în primul operand (registru).

- **BT reg/mem, im8/reg** (Bit Test)

Plasează bitul cu numărul dat de im8 sau de reg al operandului reg/mem în CF.

- **BTC reg/mem, im8/reg** (Bit Test and Complement)

Plasează bitul cu numărul dat de im8 sau de reg al operandului reg/mem în CF, apoi complementează bitul respectiv.

- **BTR reg/mem, im8/reg** (Bit Test and Reset)

Plasează bitul cu numărul dat de im8 sau de reg al operandului reg/mem în CF, apoi forțează bitul respectiv la valoarea 0.

- **BTS reg/mem, im8/reg** (Bit Test and Set)

Plasează bitul cu numărul dat de im8 sau de reg al operandului reg/mem în CF, apoi forțează bitul respectiv la valoarea 1.

instrucțiuni de control

- **JECXZ etichetă** (Jump if ECX is Zero)

Este extensia instrucțiunii JCXZ pentru registrul ECX.

- **SETccc reg/mem** (Set Bytes Conditionally)

Setează un octet conform unei condiții logice. Octetul poate fi un registru de 8 biți sau o locație de memorie. Condițiile logice și semnificațiile lor sunt aceleași ca la instrucțiunile de salt condiționat. În mnemonicele instrucțiunilor se substituie ccc cu condiția care este codificată sub forma Jccc la instrucțiunile de salt condiționat. Există deci instrucțiuni SETG, SETGE, SETNB, SETPO, SETNLE etc. De exemplu:

```
SETLE       AL
SETNBE      byte ptr [SI]
SETNC       byte ptr [EBX]
```

3.3 Coprocesoare matematice

3.3.1 Arhitectura coprocesoarelor matematice

Coprocesoarele matematice sunt circuite integrate dedicate (procesoare specializate), care extind setul de instrucțiuni al procesoarelor centrale cu instrucțiuni specifice operațiilor cu numere reale în virgulă mobilă.

Coprocesoarele sunt fie circuite de sine stătătoare (8087, 80287, 80387), fie sunt integrate în procesorul de bază (80486). În cel de-al doilea caz, nu se mai face distincție între setul de instrucțiuni al procesorului de bază și cel specific formatului în virgulă mobilă.

Tipurile de date recunoscute de coprocesoare sunt:

- număr real în simplă precizie (dword);
- număr real în dublă precizie (qword);
- număr real în precizie extinsă (tbytes);
- întreg pe 2 octeți (word);
- întreg pe 4 octeți (dword);
- întreg pe 8 biți (qword);
- întreg BCD pe 10 cifre (tbytes).

Aceste tipuri de date vor fi notate cu real32, real64, real80, int16, int32 și bcd80. Întregii pe 8 octeți pot fi numai încărcați în coprocesor. Intern, coprocesoarele lucrează exclusiv cu formatul real în precizie extinsă (10 octeți), descris în Capitolul 1. La încărcarea operanzilor din memorie, toate tipurile de date de mai sus sunt convertite la tipul intern; similar, la depunerea operanzilor în memorie, au loc conversii de la formatul intern la unul din formatele de mai sus.

Arhitectura coprocesoarelor cuprinde 8 registre de câte 80 de biți, numite ST(0), ST(1), ..., ST(7). Aceste registre sunt organizate ca o stivă, registrul ST(0) (care se mai notează simplu cu ST) fiind vârful stivei (vezi Figura 3.5). Coprocesoarele dispun de o serie de registre suplimentare, dintre care cele mai importante sunt registrul de stare (Status Word) și registrul de control (Control Word). Registrul de stare conține flaguri care se poziționează în urma instrucțiunilor de comparație sau în caz de eroare. Registrul de control conține câmpuri care controlează modul de execuție al anumitor acțiuni (de exemplu, cum se face rotunjirea la valori întregi).

Comunicația dintre procesorul de bază și coprocesor se face exclusiv prin intermediul memoriei. Coprocesorul dispune de instrucțiuni de transfer între memorie și cele 8 registre de lucru, precum și pentru cuvintele de control și de stare.

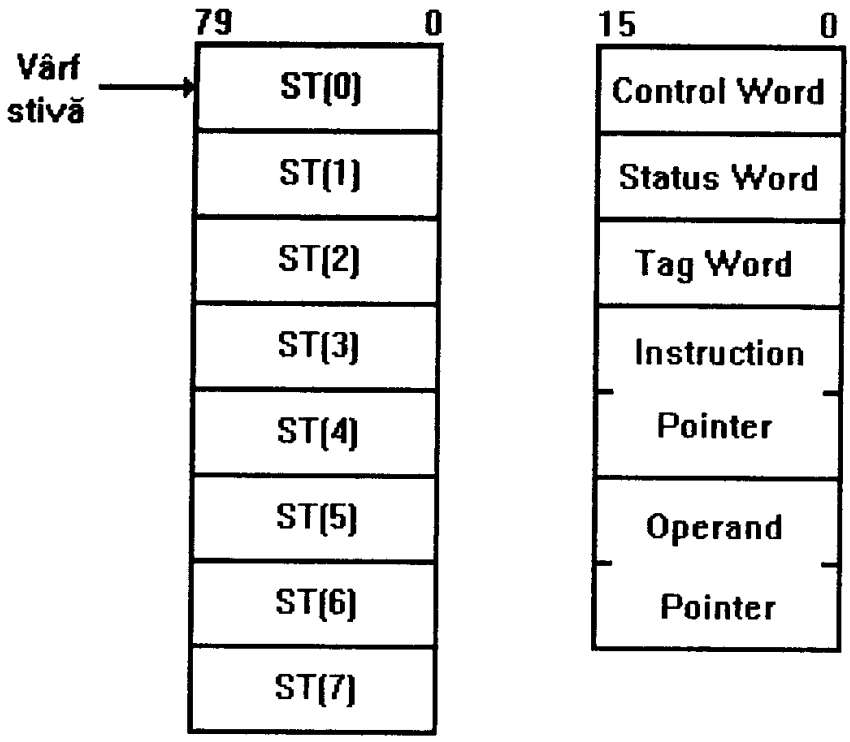


Figura 3.5 Arhitectura coprocesoarelor 80x87

Ne vom referi în continuare la coprocesorul 8087, deoarece setul de instrucțiuni este practic același la toate coprocesoarele. La procesoarele mai vechi decât 80486, se pune și problema comunicației dintre procesorul de bază și coprocesor.

Instrucțiunile specifice 8087 se scriu în textul sursă la fel ca instrucțiunile procesorului de bază. Asamblorul recunoaște mnemonicele acestor instrucțiuni și generează cod mașină corespunzător. Acest cod va fi executat de către coprocesor, în urma generării unui semnal de către procesorul de bază 8086.

Coprocesorul 8087 monitorizează permanent fluxul de instrucțiuni și sesizează prezența unei instrucțiuni specifice 8087 în memorie. În acest caz, el semnalează intenția de a intra în execuție prin semnalul electric TEST. Execuția începe numai după ce 8086 intră în așteptare, ca urmare a unei instrucțiuni WAIT. Coprocesorul recunoaște starea de așteptare și începe execuția instrucțiunii matematice, anulând în același timp cererea efectuată prin semnalul TEST. Ca urmare, 8086 iese din starea de așteptare și își continuă execuția. Asambloarele inserează automat o instrucțiune WAIT înaintea fiecărei instrucțiuni 8087, deci nu este necesară codificarea lor explicită.

La coprocesoarele de generație mai nouă (80387), sincronizarea cu procesorul de bază se face prin semnale specializate, ceea ce elimină necesitatea instrucțiunilor WAIT.

În mod corespunzător, există instrucțiunea 8087 FWAIT, destinată sincronizării reciproce (8087 „așteaptă" după 8086). Instrucțiunea FWAIT este necesară numai după operațiile de depunere în memorie executate de 8087.

Înaintea începerii operării propriu-zise, este necesară o instrucțiune FINIT, care șterge registrele interne, condițiile de eroare etc.

3.3.2 Setul de instrucțiuni 8087

Mnemonicele instrucțiunilor 8087 încep toate cu litera F. Majoritatea instrucțiunilor au ca operanzi vârful ST al stivei coprocesorului și un alt registru ST(i) sau un operand în memorie. Cele mai multe instrucțiuni matematice actualizează stiva, prin operațiile descrise mai jos (ST și ST(0) reprezintă același registru):

```
PUSH_ST:
      for (i = 7 downto 1)
            ST(i) ← ST(i-1)
POP_ST:
      for (i = 1 to 7)
            ST(i-1) ← ST(i)
```

Instrucțiuni pentru încărcarea datelor

FLD	ST(i)	(Încarcă ST(i) în ST, cu deplasarea stivei)
	temp ← ST(i)	
	PUSH_ST	
	ST ← temp	
FLD	mem (real32/64/80)	(Încarcă număr real din memorie în ST)
	PUSH_ST	
	ST ← mem	
FILD	mem (int16/32/64)	(Încarcă număr întreg din memorie în ST)
	PUSH_ST	
	ST ← mem	
FBLD	mem (bcd80)	(Încarcă număr BCD din memorie în ST)
	PUSH_ST	
	ST ← mem	
FLDZ		(Încarcă constanta 0.0 în ST)
	PUSH_ST	
	ST ← 0.0	
FLD1		(Încarcă constanta 1.0 în ST)
	PUSH_ST	

```
        ST ← 1.0
FLDPI                          (Încarcă constanta π în ST)
        PUSH_ST
        ST ← π
FLDL2E                         (Încarcă constanta log2(e) în ST)
        PUSH_ST
        ST ← log2(e)
FLDL2T                         (Încarcă constanta log2(10) în ST)
        PUSH_ST
        ST ← log2(10)
FLDLG2                         (Încarcă constanta log10(2) în ST)
        PUSH_ST
        ST ← log10(2)
```

Instrucțiuni pentru depunerea datelor

```
FST     ST(i)                  (Depune ST în ST(i) fără descărcarea stivei)
        ST(i) ← ST
FSTP    ST(i)                  (Depune ST în ST(i) cu descărcarea stivei)
        ST(i) ← ST
        POP_ST
FST     mem (real32/64/80)     (Depune ST real în memorie fără descărcare)
        mem ← ST
FSTP    mem (real32/64/80)     (Depune ST real în memorie cu descărcare)
        mem ← ST
        POP_ST
FIST    mem (int16/32)         (Depune ST întreg în memorie fără descărcare)
        mem ← ST
FISTP   mem (int16/32)         (Depune ST întreg în memorie cu descărcare)
        mem ← ST
        POP_ST
FBSTP   mem (bcd80)            (Depune ST BCD în memorie cu descărcare)
        mem ← ST
        POP_ST
FXCH                           (Schimbă ST cu ST(1))
```

```
FXCH  ST(i)                    (Schimbă ST cu ST(i))
```

Instrucțiuni de adunare

```
FADD                           (Adună ST cu ST(1), cu descărcare)
      ST(1) ← ST(1) + ST
      POP_ST
FADD  ST(i), ST                (Adună ST la ST(i))
      ST(i) ← ST(i) + ST
FADD  ST, ST(i)                (Adună ST(i) la ST)
      ST ← ST + ST(i)
FADD  mem (real32/64/80)       (Adună real din memorie la ST)
      ST ← ST + mem
FIADD mem (int16/32)           (Adună întreg din memorie la ST)
      ST ← ST + mem
FADDP ST(i)                    (Adună ST la ST(i) cu descărcarea stivei)
      ST(i) ← ST(i) + ST
      POP_ST
```

Instrucțiuni de scădere

```
FSUB                           (Calculează ST(1) - ST cu descărcare)
      ST(1) ← ST(1) - ST
      POP_ST
FSUB  ST(i), ST                (Scade ST din ST(i))
      ST(i) ← ST(i) - ST
FSUB  ST, ST(i)                (Scade ST(i) din ST)
      ST ← ST - ST(i)
FSUB  mem (real32/64/80)       (Scade real în memorie din ST)
      ST ← ST - mem
FISUB mem (int16/32)           (Scade întreg în memorie din ST)
      ST ← ST - mem
FSUBP ST(i)                    (Scade ST din ST(i) cu descărcare)
      ST(i) ← ST(i) - ST
      POP_ST
FSUBR                          (Calculează ST - ST(1) cu descărcare)
      temp ← ST - ST(1)
      POP_ST
```

```
        ST ← temp
FSUBR   ST(i), ST           (Scade ST(i) din ST, rezultat în ST(i))
        ST(i) ← ST - ST(i)
FSUBR   ST, ST(i)           (Scade ST din ST(i), rezultat în ST)
        ST ← ST(i) - ST
FSUBR   mem (real32/64/80)  (Scade ST din real în memorie, rezultat în ST)
        ST ← mem - ST
FISUBR  mem (int16/32)      (Scade ST din întreg în memorie, rezultat în ST)
        ST ← mem - ST
FSUBPR  ST(i)               (Scade ST(i) din ST cu descărcare)
        ST(i) ← ST - ST(i)
        POP_ST
```

Instrucțiuni de înmulțire

```
FMUL                        (Înmulțește ST cu ST(1), cu descărcare)
        ST(1) ← ST(1) * ST
        POP_ST
FMUL    ST(i), ST           (Înmulțește ST cu ST(i), rezultat în ST(i))
        ST(i) ← ST(i) * ST
FMUL    ST, ST(i)           (Înmulțește ST(i) cu ST, rezultat în ST(i))
        ST ← ST * ST(i)
FMUL    mem (real32/64/80)  (Înmulțește real din memorie cu ST)
        ST ← ST * mem
FIMUL   mem (int16/32)      (Înmulțește întreg din memorie cu ST)
        ST ← ST * mem
FMULP   ST(i)               (Înmulțește ST cu ST(i), cu descărcare)
        ST(i) ← ST(i) * ST
        POP_ST
```

Instrucțiuni de împărțire

```
FDIV                        (Calculează ST(1) / ST cu descărcare)
        ST(1) ← ST(1) / ST
        POP_ST
FDIV    ST(i), ST           (Împarte ST(i) la ST, rezultat în ST)
        ST(i) ← ST(i) / ST
FDIV    ST, ST(i)           (Împarte ST la ST(i), rezultat în ST(i))
```

```
        ST ← ST / ST(i)
FDIV    mem (real32/64/80)      (Împarte ST la real din memorie)
        ST ← ST / mem
FIDIV   mem (int16/32)          (Împarte ST la întreg din memorie)
        ST ← ST / mem
FDIVP   ST(i)                   (Împarte ST(i) la ST, cu descărcare)
        ST(i) ← ST(i) / ST
        POP_ST
FDIVR                           (Calculează ST / ST(1), cu descărcare)
        temp ← ST / ST(1)
        POP_ST
        ST ← temp
FDIVR   ST(i), ST               (Împarte ST la ST(i), rezultat în ST(i))
        ST(i) ← ST / ST(i)
FDIVR   ST, ST(i)               (Împarte ST(i) la ST, rezultat în ST)
        ST ← ST(i) / ST
FDIVR   mem (real32/64/80)      (Împarte real în memorie la ST, rezultat în ST)
        ST ← mem / ST
FIDIVR  mem (int16/32)          (Împarte întreg în memorie la ST, rezultat în ST)
        ST ← mem / ST
FDIVPR  ST(i)                   (Împarte ST la ST(i), cu descărcare)
        ST(i) ← ST / ST(i)
        POP_ST
```

Instrucțiuni de comparație

Instrucțiunile de comparație poziționează indicatorii din cuvântul de stare prin efectuarea unei operații temporare de scădere între operanzii care se compară.

```
FCOM                            (Compară ST cu ST(1))
FCOM    ST(i)                   (Compară ST cu ST(i)
FCOM    mem (real32/64/80)      (Compară ST cu real din memorie)
FICOM   mem (int16/32)          (Compară ST cu întreg din memorie)
FTST                            (Compară ST cu 0.0)
FCOMP                           (Compară ST cu ST(1), cu descărcare)
        temp ← ST - ST(1)
        POP_ST
```

FCOMP	ST(i)	(Compară ST cu ST(i), cu descărcare)
FCOMP	mem (real32/64/80)	(Compară ST cu real, cu descărcare)
FICOMP	mem (int16/32)	(Compară ST cu întreg, cu descărcare)
FCOMPP		(Compară ST cu ST(1), cu descărcare dublă)

```
temp ← ST - ST(1)
POP_ST
POP_ST
```

Instrucțiuni diverse

FABS		(la valoarea absolută)

```
ST ← | ST |
```

FCHS		(Schimbă semnul)

```
ST ← - ST
```

FSQRT		(Radical)

```
ST ← sqrt (ST)
```

Instrucțiuni de control

FINIT		(Inițializează coprocesor)
FWAIT		(Sincronizează cu procesorul de bază)
FSTSW	mem (int16)	(Depune cuvânt de stare în memorie)
FSTCW	mem (int16)	(Depune cuvânt de control în memorie)
FLDCW	mem (int16)	(Încarcă cuvânt de control din memorie)
FNOP		(Nici o operație)

Pe lângă instrucțiunile de mai sus, coprocesoarele dispun de instrucțiuni transcendente, pentru calculul funcțiilor trigonometrice directe și inverse, exponențială, logaritm etc. Aceste instrucțiuni nu vor fi prezentate, deoarece este puțin probabil că se vor dezvolta programe cu funcții transcendente în limbaj de asamblare.

În Capitolul 6, vor fi prezentate tehnici de programare specifice coprocesoarelor matematice.

Capitolul 4

Structurarea programelor
Definirea și inițializarea datelor
Operatori

Capitolul curent și cel următor sunt dedicate limbajului de asamblare. Pe lângă instrucțiunile propriu-zise, un program ASM poate cuprinde directive, prin intermediul cărora se definesc date, etichete și proceduri, se structurează segmente, se definesc și se utilizează macroinstrucțiuni, se controlează în general procesul de asamblare etc. Directivele nu reprezintă instrucțiuni, ci comenzi către asamblor, efectul lor manifestându-se exclusiv în faza de asamblare.

4.1 Segmentare. Directive pentru definirea segmentelor

Un modul de program în limbajul de asamblare poate conduce la:

- o porțiune dintr-un segment;
- un segment;
- porțiuni de segmente diferite;
- mai multe segmente.

Mai multe module obiect se leagă prin operația de link-editare.

4.1.1 Directivele SEGMENT și ENDS

Indiferent de modul de dezvoltare al unui program ASM, atât instrucțiunile, cât și datele trebuie să se găsească în interiorul unui segment. Directiva SEGMENT controlează:

- numele segmentului;
- alinierea;
- combinarea cu alte segmente;
- continuitatea (adiacența) segmentelor.

Forma generală a directivei SEGMENT este:

```
nume SEGMENT [tip_aliniere] [tip_combinare] ['nume_clasa']
        .
        .
nume ENDS
```

Parametrii incluși în paranteze drepte sunt opționali. Dacă sunt prezenți, aceștia trebuie să fie în ordinea specificată. Semnificația parametrilor este:

- **tip_aliniere** - specifică la ce limită va fi relocat segmentul în memorie; poate avea una din formele:
  - PARA (implicit) - aliniere la paragraf (segmentul fizic, adică adresa pe 20 de biți, va fi relocat la prima adresă absolută divizibilă prin 16);
  - BYTE - fără aliniere (segmentul se va reloca la următorul octet liber);
  - WORD - aliniere la cuvânt (segmentul se va reloca la următorul octet liber aflat la adresă pară);
  - DWORD - aliniere la dublu-cuvânt (segmentul se va reloca la prima adresă divizibilă cu 4);
  - PAGE - aliniere la pagină (segmentul se va reloca la prima adresă absolută divizibilă prin 256).

Să considerăm următoarele definiții de segmente:

```
DAT1 SEGMENT BYTE
      x1     db 7 dup (?)
DAT1 ENDS
DAT2 SEGMENT WORD
      x21    dw 300 dup (?)
      x22    dw ?
DAT2 ENDS
DAT3 SEGMENT PARA
       x3    db 5 dup (?)
DAT3 ENDS
```

Dacă prima adresă disponibilă este 1000H, cele trei segmente vor fi relocate la următoarele adrese fizice:

DAT1 1000:0 ÷ 1000:6

DAT2 1000:8 ÷ 1000:261

DAT3 1027:0 ÷ 1027:4

Adresele de segment se obțin prin trunchierea la primele patru cifre ale adresei fizice de început, iar offset-urile se calculează corespunzător. Offset-urile la execuție diferă în general de offset-urile la asamblare. Offset-ul la asamblare pentru x21 este 0, dar la execuție este 8.

Operatorul OFFSET, care furnizează deplasamentul unei variabile sau unei etichete în cadrul unui segment, va produce offset-ul de la execuție. Instrucțiunea:

```
mov   bx, OFFSET x21
```

va încărca în BX valoarea 8.

Dacă dorim ca offset-ul la asamblare să coincidă cu offset-ul la execuție, putem folosi tipul de aliniere PARA (ultima cifră a adresei fizice de început a segmentului este 0).

- **tip_combinare** - specifică dacă și cum se va combina segmentul respectiv cu alte segmente, la operația de link-editare; poate fi:
  - necombinabil (implicit) - nu se scrie nimic;
  - PUBLIC - segmentul curent va fi concatenat cu alte segmente cu același nume și cu atributul PUBLIC. Aceste segmente pot fi în alte module de program. Se va forma, deci, un singur segment final cu numele respectiv, cu o unică adresă de început; lungimea unui segment cu atributul PUBLIC este suma lungimilor segmentelor cu același nume.
  - COMMON - specifică faptul că segmentul curent și toate segmentele cu același nume și tipul COMMON se vor suprapune în memorie (încep la aceeași adresă fizică); lungimea unui segment COMMON este cea mai mare lungime a segmentelor componente;
  - STACK - marchează segmentul curent ca reprezentând stiva programului; dacă sunt mai multe segmente cu tipul STACK, ele vor fi tratate ca la PUBLIC; următorul exemplu este o definire și o inițializare explicită a unui segment de stivă:

```
stiva       SEGMENT STACK
            db    256 dup(?)
            stiva_sp label WORD
stiva       ENDS
cod   SEGMENT
            mov   ax, stiva
            mov   ss, ax
            mov   sp, OFFSET stiva_sp
cod   ENDS
```

  - AT EXPRESIE - specifică faptul că segmentul va fi plasat la o adresă fizică absolută de memorie; următorul exemplu ilustrează definirea explicită a tabelei de vectori de întreruperi:

```
INT_TABLE SEGMENT AT 0
      dd PROC_NIV_0
      dd PROC_NIV_1
            .
            .
INT_TABLE ENDS
```

Aceste forme se folosesc când programul este dezvoltat pentru echipamente dedicate. În cazul unui calculator de tip IBM-PC, există funcții DOS pentru accesul la tabela de întreruperi.

- **'nume_clasă'** - specifică un nume de clasă pentru segment, extinzând astfel numele segmentului; de exemplu, dacă segmentul are și atributul 'nume_clasă', atunci atributele COMMON sau PUBLIC vor acționa numai asupra segmentelor cu același nume și același nume de clasă; ca nume de clase, se folosesc de obicei 'code', 'data' și 'stack'.

Să considerăm două module ASM, asamblate distinct. În primul modul, avem următoarele definiții de segmente:

```
dseg1 SEGMENT PARA PUBLIC 'data'
      db    100 dup (?)
dseg1 ENDS
dseg2 SEGMENT PARA COMMON
      db 50 dup (?)
dseg2 ENDS
cseg  SEGMENT PARA PUBLIC 'code'
      db    20 dup (?)
cseg  ENDS
```

În al doilea modul, există următoarele definiții:

```
dseg1 SEGMENT PARA PUBLIC 'data'
      db 30 dup (?)
dseg1 ENDS
dseg2 SEGMENT PARA COMMON
      db    200 dup (?)
dseg2 ENDS
cseg  SEGMENT PARA PUBLIC 'code'
      db    70 dup (?)
cseg  ENDS
```

Imaginea segmentelor rezultate în urma link-editării și a relocării este cea din Figura 4.1.

4.1.2 Directiva ASSUME

Directiva ASSUME realizează o conexiune simbolică (logică) între definirea instrucțiunilor și a datelor în segmente logice (cuprinse între SEGMENT și ENDS) la momentul asamblării și accesul, la execuție, la instrucțiuni și date prin registrele de segment. Are forma generală:

```
ASSUME reg_seg:expresie, ...
```

în care reg_seg este un registru de segment, iar expresie poate fi;

- un nume de segment;
- un nume de grup de segmente;
- SEG nume segment;
- NOTHING.

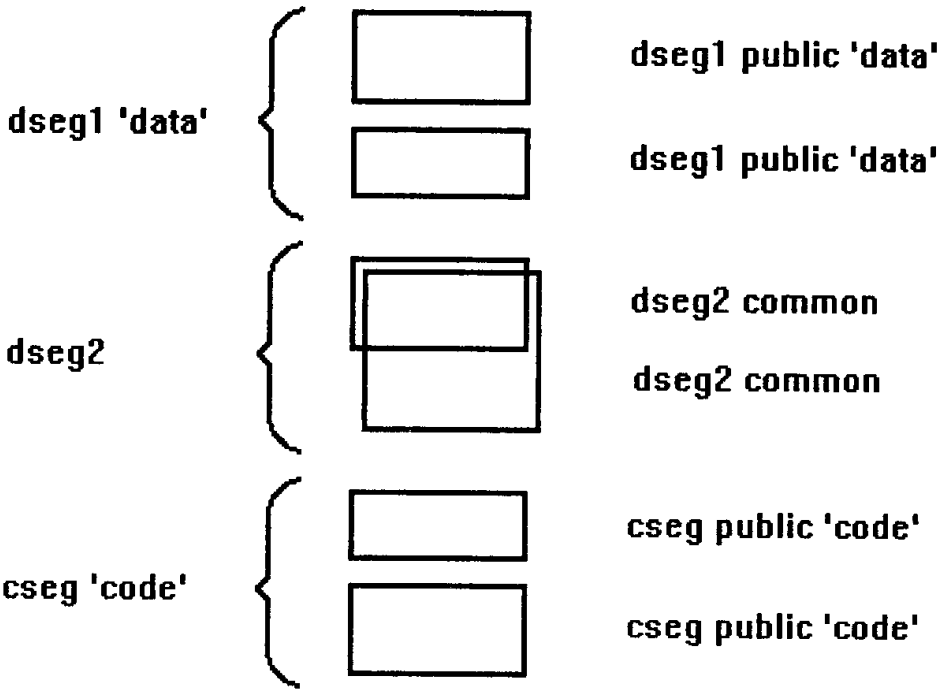


Figura 4.1 Combinarea segmentelor este controlată de directiva SEGMENT

În aceeași directivă ASSUME se pot asocia mai multe registre de segment, de exemplu:

```
ASSUME CS:CSEG, DS:DSEG1, ES:DSEG2, SS:SSEG
```

Directiva ASSUME nu ne scutește de încărcarea registrelor de segment cu adresele de segment. Dacă un nume de segment nu apare într-o directivă ASSUME, datele definite în acel segment pot fi accesate numai prin prefixe de segment. Să considerăm următorul exemplu:

```
dseg   SEGMENT
       BETA  dw 5
dseg   ENDS
cseg   SEGMENT
       ASSUME cs:cseg, es:dseg
       mov   ax, dseg
       mov   es, ax
       mov   bx, BETA      ; se va genera o instructiune MOV,
                           ; cu adresare prin registrul ES
                           ; conform asocierii ES:dseg
cseg   ENDS
```

Dacă nu s-ar fi asociat registrul ES cu segmentul dseg prin directiva ASSUME, accesul la variabila BETA trebuia scris în forma:

```
mov bx, es:BETA
```

În formatul intern al instrucțiunii (în codificare), trebuie să apară prin ce registru de segment se calculează adresa fizică. Dacă simbolul care apare într-o instrucțiune (de exemplu, BETA), face parte dintr-un segment care este asociat, printr-o directivă ASSUME, cu un registru de segment, atunci se va codifica intern accesul prin acel registru de segment (în cazul de față, prin ES). Codificarea registrului de segment se face:

- **implicit** - datele sunt accesate în toate modurile de adresare prin registrul DS, cu excepția celor care implică registrul BP, caz în care se folosește SS, iar instrucțiunile sunt adresate totdeauna prin CS;
- **explicit** - prin utilizarea prefixelor de segment.

4.1.3 Directiva GROUP

Directiva GROUP servește la gruparea mai multor segmente (inclusiv cu nume și atribute diferite), a căror lungime totală nu depășește 64 KO, sub același nume. Are forma generală:

```
nume_grup GROUP nume_seg1, nume_seg2, ...
```

Această directivă reprezintă o altă modalitate de combinare a mai multor segmente, pe lângă cea oferită de atributul PUBLIC.

Numele de grup se poate folosi în directive ASSUME, la inițializarea unor registre de segment sau la calculul unui offset. Să considerăm următorul exemplu:

```
d1      SEGMENT
              y       dw      20
d1      ENDS
d2      SEGMENT
              x       db      2
d2      ENDS
data    GROUP         d1, d2
```

Putem scrie instrucțiuni de forma:

```
mov     ax, data
mov     ds, ax
mov     ax, OFFSET x
mov     ax, OFFSET data:x
```

sau directive de forma:

```
ASSUME        ds:data
```

Expresia OFFSET x furnizează offset-ul variabilei x în cadrul segmentului d1, iar expresia OFFSET data:x produce offset-ul variabilei x în cadrul grupului de segmente data.

4.2 Definirea simplificată a segmentelor

Această modalitate de definire este introdusă în variantele recente ale asambloarelor. Avantajul major este faptul că se respectă același format (structură a programului obiect) ca și la programele dezvoltate în limbaj de nivel înalt. Concret, dacă utilizăm definirea simplificată, se vor genera segmente cu nume și atribute identice cu cele generate de compilatoarele de limbaje de nivel înalt. Toate directivele simplificate încep cu un punct.

4.2.1 Modele de memorie

Directiva pentru specificarea modelului de memorie are forma generală:

.model tip

în care tip poate fi **tiny**, **small**, **medium**, **large** sau **huge**. Semnificația acestor tipuri este:

- **tiny** - toate segmentele (date, cod, stivă) se pot genera într-un spațiu de 64KO și formează un singur grup de segmente. Se folosește la programele de tip COM. Toate salturile, apelurile și definițiile de proceduri sunt implicit de tip NEAR;
- **small** - datele și stiva sunt grupate într-un singur segment, iar codul în alt segment. Fiecare din acestea nu trebuie să depășească 64KO. Toate salturile, apelurile și definițiile de proceduri sunt implicit de tip NEAR;
- **medium** - datele și stiva sunt grupate într-un singur segment (cel mult egal cu 64KO), dar codul poate fi în mai multe segmente separate (nu se grupează), deci poate depăși 64KO. Salturile și apelurile sunt implicit tip FAR, iar definițiile de proceduri sunt implicit de tip far.
- **compact** - codul generat ocupă cel mult 64KO (se grupează), dar datele și stiva sunt în segmente separate (pot depăși 64KO). Apelurile și salturile sunt implicit de tip NEAR. Se utilizează adrese complete (segment și offset) atunci când se accesează date definite în alte segmente.
- **large** - atât datele, cât și codul generat pot depăși 64KO.
- **huge** - este asemănător modelului large, dar structurile de date pot depăși 64KO; se utilizează adrese complete normalizate în care offset-ul este redus la minim (în domeniul 0-15), ceea ce face ca o adresă fizică să fie descrisă de o unică pereche (segment, offset).

La modelele compact și large, o structură compactă de date (de tip tablou) nu poate depăși limitele unui segment fizic (64KO); la modelul huge, această restricție dispare.

Folosirea directivelor simplificate prezintă și avantajul că nu mai sunt necesare directive ASSUME: asocierile între segmentele generate și registrele de segment sunt implicite.

Se utilizează următoarea terminologie:

- modele de date mici: small, compact;
- modele de cod mic: small, medium;
- modele de date mari: medium, large, huge;
- modele de cod mare: compact, large, huge.

4.2.2 Directive simplificate de definire a segmentelor

Formele generale ale acestor directive sunt:

- .stack dimensiune
- .code [nume]
- .data
- .data? ; date neinițializate în l.n.i
- .fardata [nume] ; segmente de date utilizate prin
- .fardata? [nume] ; adrese complete în l.n.i
- .const ; definire de constante în l.n.i)

Dacă parametrul nume lipsește, se atribuie nume implicite segmentelor generate, după cum urmează:

- pentru .code TEXT (la modele de cod mic) sau nume_fișier_sursă_TEXT (la modele de cod mare);
- pentru .data? _BSS;
- pentru .data DATA;
- pentru .const CONST
- pentru .stack STACK
- pentru .fardata _FAR_DATA
- pentru .fardata? _FAR_BSS

Se generează grupul de segmente DGROUP. Acesta cuprinde:

- toate segmentele, la modelul de date tiny;
- DATA, _BSS, CONST, STACK, în rest.

Se consideră că există o directivă ASSUME implicită (care nu mai trebuie, deci, scrisă în textul sursă), de forma:

- la modelele small și compact:

```
ASSUME cs:TEXT, ds:DGROUP, ss:DGROUP
```

- la modelele large și huge:

```
ASSUME cs:nume_TEXT, ds:DGROUP, ss:DGROUP
; nume este numele din directiva .code
; sau numele fisierului sursa
```

- la modelul de memorie tiny:

```
ASSUME cs:DGROUP, ds:DGROUP, cs:DGROUP
```

Dacă se folosesc directivele .fardata și/sau .fardata?, atunci segmentele respective trebuie gestionate explicit prin directive ASSUME.

Adresele de început ale segmentelor și ale grupurilor de segmente definite cu directivele simplificate sunt disponibile prin simbolurile globale:

@data, @data?, @fardata, @fardata?, dgroup.

Dacă în același text sursă (model de cod mare) se folosesc mai multe directive .code cu nume diferite, atunci, până la întâlnirea unei alte directive de segment, este ca și cum s-ar fi scris o directivă ASSUME CS: cu numele respectiv.

Segmentele generate au numele de clasă 'STACK', 'CODE', 'DATA', 'BSS', 'FAR_BSS', 'FAR_DATA'. Segmentul CONST are numele de clasă DATA.

Tipurile de combinare sunt:

- PUBLIC (la .code, .data, .data? și .const);
- STACK (la .stack);
- fără combinare (la .fardata și .fardata?).

Tipul de aliniere este PARA (pentru .stack, .fardata și .fardata?) și WORD (pentru celelalte).

Să considerăm fișierul sursă **exemplu.asm**, cu conținutul de mai jos:

```
.model      large
.data
.const
.stack      256
.data?
.fardata
.fardata?
.code
end
```

Fișierul listing generat cuprinde (între altele), următoarele informații:

```
SymbolName  Type        Value
??DATE      Text        "06/02/96"
??FILENAME  Text        "exemplu"
??TIME      Text        "15:21:50"
??VERSION   Number      0200
@CODE       Text        EXEMPLU_TEXT
@CODESIZE   Text        1
@CPU        Text        0101H
@CURSEG     Text        EXEMPLU_TEXT
@DATA       Text        DGROUP
@DATASIZE   Text        1
@FARDATA    Text        FAR_DATA
@FARDATA?   Text         FAR_BSS
@FILENAME   Text        EXEMPLU
@MODEL      Text         5
```

```
@WORDSIZE   Text          2

Groups & Segments       Bit   Size  Align Combine Class
DGROUP Group
      CONST             16    0000  Word  Public DATA
      STACK             16    0100  Para  Stack  STACK
       _BSS             16    0000  Word  Public BSS
      _DATA             16    0000  Word  Public DATA
      EXEMPLU_TEXT      16    0000  Word  Public CODE
      FAR_BSS           16    0000  Para  none   FAR_BSS
      FAR_DATA          16    0000  Para  none   FAR_DATA
```

Listingul relevă o serie de simboluri predefinite și valorile asociate. Se remarcă adresele de început ale segmentelor, cel mai important fiind simbolul @DATA, care are aceeași valoare cu adresa grupului DGROUP. Partea a doua a listingului descrie segmentele și grupurile de segmente.

Inițializarea registrelor DS și ES la începutul unui program principal se poate face cu simbolul @DATA sau cu simbolul DGROUP. Macroinstrucțiunea init_ds_es realizează operația:

```
mov   ax, dgroup
mov   ds, ax
mov   es, ax
```

4.3 Directive pentru legarea modulelor

Când programul dezvoltat se compune din mai multe module asamblate separat, este necesară specificarea simbolurilor care sunt definite într-un modul și utilizate în alte module. Aceste simboluri sunt nume de variabile, etichete sau nume de proceduri. În mod normal, un simbol este vizibil doar în modulul în care este definit (simbol local). Simbolurile care sunt vizibile în mai multe module de program se numesc simboluri globale.

Se folosesc următoarele noțiuni:

- simboluri **publice** - sunt simboluri care sunt definite într-un modul curent de program și folosite (eventual) și în alte module; aceste simboluri se declară ca simboluri publice în modulul în care sunt definite;
- simboluri **externe** - sunt simboluri care sunt folosite într-unul din modulele în care nu sunt definite; aceste simboluri se declară ca simboluri externe în modulele în care sunt folosite.

Se observă că un simbol global trebuie declarat ca simbol public în modulul în care este definit și ca simbol extern în modulele în care este folosit, altele decât cel în care este definit.

Declarația unui simbol ca simbol public, respectiv extern se face cu directivele PUBLIC, respectiv EXTRN.

Directiva PUBLIC are forma generală:

```
PUBLIC nume, nume, ...
```

Simbolurile declarate ca publice pot fi variabile, etichete, nume de proceduri sau constante numerice simbolice.

Directiva EXTRN are forma generală:

```
EXTRN nume:tip, nume:tip, ...
```

în care tip precizează tipul simbolului. Acesta poate fi:

- BYTE, WORD, DWORD sau QWORD, în cazul în care simbolul este o variabilă;
- NEAR sau FAR, în cazul în care simbolul este o etichetă sau un nume de procedură;
- ABS, în cazul în care simbolul este o constantă numerică simbolică.

Să considerăm două module de program, M1.ASM și M2.ASM:

```
; Modul M1.ASM
;
.model large
      extrn       var1:word, output:far
      public      var2
.data
      var2        dw 7
.code
start:
      mov         ax, dgroup
      mov         ds, ax
      add         var1, 3
      call        output
end start

; Modul M2.ASM
;
.model large
      public      var1, output
      extrn       var2:word
.data
      var 1       dw5
.code
      output proc far
                  add var2, 1
                  retf
      output endp
end
```

În acest exemplu, accesul la simbolurile globale este facilitat de faptul că atât var1, cât și var2 sunt în același segment de date, fiind accesibile prin registrul DS, inițializat cu grupul de segmente DGROUP.

Dacă nu am fi folosit directive simplificate, accesul la var1 și var2 ar fi fost mai complicat, deoarece nu s-ar fi cunoscut segmentul în care acestea sunt definite. În asemenea situații, se folosește operatorul SEG, care produce adresa de segment a simbolului:

```
mov ax, SEG var1
mov es, ax
add es:var1, 3
```

Directiva END

Această directivă marchează sfârșitul logic al unui modul de program și e obligatorie în toate modulele. Tot ce se găsește în fișierul sursă după această directivă este ignorat la asamblare. Forma generală este:

```
END [punct_de_start]
```

în care punct_de_start este o etichetă sau un nume de procedură care marchează punctul în care se va da controlul după încărcarea programului în memorie. Într-o aplicație compusă din mai multe module și care se constituie într-un program executabil, un singur modul trebuie să aibă parametru în directiva END.

4.4 Contoare de locații și directiva ORG

Contoarele de locații controlează procesul de asamblare, arătând la ce offset în cadrul segmentului curent se vor asambla instrucțiunea sau datele următoare. Un contor de locații poate fi accesat explicit prin simbolul $.

La prima utilizare a unui nume de segment, contorul de locații este inițializat cu zero. Dacă se revine într-un segment care a mai fost utilizat, contorul de locații revine la ultima valoare utilizată în cadrul acelui segment, ca în exemplul următor:

```
.data                   ; $ = 0
 db 5 dup(?)
                        ; $ = 5
.code
                        ; $ = 0
.data
 db 7                   ; $ = 5
```

Contoarele de locații sunt utile la calculul automat al unor deplasamente sau dimensiuni. În exemplul următor, se definește un tablou de cuvinte TAB și o variabilă LNG care conține numărul de cuvinte din tablou:

```
TAB dw 1, 2 , 3, 4, 5, 6, 7, 8, 9
LNG dw ($ - TAB)/2
```

Expresia $ - TAB reprezintă numărul de octeți de la adresa TAB până la adresa curentă. Împărțind acest număr la 2, se deduce numărul de elemente din tablou. Putem acum adăuga sau elimina date din tabloul TAB fără a actualiza lungimea LNG: asamblorul va calcula corect numărul de elemente din tablou.

Alte exemple de utilizare pot fi:

```
JMP $        ; Ciclu infinit: $ este chiar adresa
             ; instructiunii JMP
JMP $ + 5    ; salt relativ
```

Directiva ORG (Origin - Inițializează contorul de locații)

Această directivă modifică explicit contorul de locații curent, având forma generală:

```
ORG expresie
```

De exemplu:

```
ORG $ + 5 ; Sare 5 octeti la asamblare
ORG 100H ; Asambleaza la offset-ul absolut 100H
```

4.5 Definirea și inițializarea datelor

Asamblorul recunoaște trei categorii sintactice de bază:

- constante;
- variabile;
- etichete (inclusiv nume de proceduri).

Constantele pot fi absolute (numerice) sau simbolice. Constantele simbolice reprezintă nume generice asociate unor valori numerice. Variabilele identifică datele (un spațiu de memorie rezervat), iar etichetele identifică codul (program sau procedură).

Instrucțiuni de tipul MOV, ADD, AND etc. folosesc variabile și constante, iar instrucțiunile JMP și CALL folosesc etichete.

Variabilele și etichetele au asociate atribute, cum ar fi segmentul în care sunt definite, offset-ul la care sunt definite în interiorul segmentului etc.

4.5.1 Constante

Constantele numerice absolute pot fi:

- constante binare - se utilizează sufixul B sau b;
- constante octale - se utilizează sufixele O, Q, o sau q;
- constante zecimale - fără sufix sau cu sufixele D sau d;
- constante hexazecimale - se utilizează sufixele H sau h și prefixul 0 dacă prima cifră este mai mare ca 9; pentru cifrele peste 9 se utilizează simbolurile A...F sau a...f.

- constante ASCII - se scriu unul sau mai multe caractere ASCII între semnul apostrof sau ghilimele;

Iată exemple de constante absolute: 123, 123D, 10001100B, 177q, 0AAH, 3fh, 'a', "AB".

Constantele simbolice se definesc cu directiva EQU, în forma generală:

```
nume EQU expresie
```

De exemplu, liniile de program:

```
CR EQU 0DH
LF EQU 0AH
```

definesc constantele simbolice CR și LF, cu valorile 0DH și 0AH. Putem folosi aceste constante simbolice în orice context în care este permisă folosirea unei constante numerice.

4.5.2 Definirea și utilizarea variabilelor

Pentru definirea variabilelor, se utilizează directivele DB, DW, DD, DQ sau DT, care au fost introduse în Capitolul 1. Forma generală a unei definiții de date este:

```
nume directiva lista_de_valori
```

în care nume este identificatorul asociat definiției respective, iar listă_de_valori este lista valorilor inițiale, care poate cuprinde:

- constante numerice (absolute sau simbolice);
- simbolul ?
- o adresă, adică un nume de variabilă sau de etichetă; se folosește la DW și DD;
- un șir ASCII;
- operatorul DUP (expresie), în care expresie poate fi:
  - ♦ constantă numerică;
  - ♦ listă de valori;
  - ♦ simbolul ?
  - ♦ operatorul DUP.

Semnificația simbolului ? este locație neinițializată, iar cea a operatorului DUP, repetarea de un număr de ori a expresiei din paranteză.

Să considerăm definițiile următoare:

```
var   a     db 2 dup (0, 3 dup (1))
var_  b     db 1, 2, 3, ?, ?, ?
adr_  n     dw var_a
adr_  f     dd var_b
```

Prima definiție este echivalentă cu:

```
var_a db 0, 1, 1, 1, 0, 1, 1, 1
```

Atributele datelor definite sunt:

- segment - cel curent;
- offset - cel curent;
- tip - 1, 2, 4, 8, 10, după cum s-a utilizat ca directivă de bază DB, DW, DD, DQ sau DT.

Variabilele pot fi utilizate ca:

- variabile simple - var_a, var_b;
- variabile indexate - var_a [bx], var_b [2], ALFA [si][bx].

În cazul variabilelor indexate, trebuie ținut seama de tipul variabilei de bază. De exemplu, în urma definiției tabloului:

```
B dw 10 dup(?)
```

expresiile indexate corecte din punct de vedere logic, care accesează elementele lui B, sunt B[0], B[2] etc. Din punct de vedere sintactic, putem scrie și:

```
mov ax, B [1]
```

dar această formă va încărca în AL partea high a primului cuvânt din tablou și în AH partea low a celui de-al doilea cuvânt din tabloul B. Dacă acest lucru s-a urmărit, perfect !

În cazul accesării variabilelor prin expresii anonime, poate apărea necesitatea operatorului PTR. Instrucțiunile:

```
lea bx, B
inc byte ptr [bx][4]
inc word ptr [bx][4]
```

incrementează octetul, respectiv cuvântul de la adresa BX + 4.

4.6 Definirea etichetelor. Directiva LABEL

Etichetele sunt folosite pentru specificarea punctelor țintă la instrucțiuni de salt sau apel sau pentru o specificare alternativă a datelor. În ambele cazuri, numele etichetei devine un nume simbolic asociat adresei curente de memorie. Atributele etichetelor sunt: segment, offset și tip. Până acum am întâlnit două modalități de definire a etichetelor:

- prin nume urmat de caracterul : - se definește o etichetă de tip near;
- prin directiva PROC - numele procedurii este interpretat ca o etichetă cu tipul derivat din tipul procedurii;

O altă posibilitate este **directiva LABEL**, care are forma generală:

```
nume LABEL tip
```

Dacă ceea ce urmează reprezintă instrucțiuni (cod), tipul etichetei va fi, de regulă, NEAR sau FAR și eticheta va fi folosită ca punct țintă în instrucțiuni de timp JMP/CALL. Dacă ceea ce urmează reprezintă definiții de date, tipul etichetei va fi, de regulă, BYTE, WORD, DWORD etc.

Atributele NEAR sau FAR pot apărea numai dacă eticheta este definită într-un segment asociat cu registrul CS printr-o directivă ASSUME (explicită sau implicită).

Directiva LABEL este utilă atunci când dorim să accesăm variabile în moduri diferite. De exemplu, în urma definiției:

```
ALFAB label BYTE
ALFAW dw 1234H
```

o instrucțiune de forma inc ALFAB va incrementa octetul mai puțin semnificativ al cuvântului, iar una de tip inc ALFAW va incrementa întreg cuvântul; un efect similar se poate obține prin operatorul PTR.

Similar, definiția:

```
adr_proc label dword
proc_off dw ?
proc_seg dw ?
```

permite definirea explicită a segmentului și a offset-ului, dar și accesul la nivel de dublu-cuvânt, de exemplu într-o instrucțiune de apel indirect inter-segment.

4.7 Definirea structurilor. Operații specifice

Structurile reprezintă colecții de date (câmpuri sau membri) plasate succesiv în memorie, grupate sub un unic nume sintactic. Dimensiunea unei structuri este suma dimensiunilor câmpurilor componente. Forma generală a definiției unei structuri este:

```
nume_structura STRUC
  .
  .
nume_membru definitie_date
  .
  .
nume_structura ENDS
```

Prin aceasta se definește tipul structurii (șablonul) fără a se rezerva spațiu de memorie. Definițiile de date pot cuprinde directive DB, DW etc. cu valori inițiale asociate, ?, DUP etc., exact ca orice definiție de date. Numele membrilor structurii trebuie să fie distincte, chiar dacă aparțin unor structuri distincte. Iată un exemplu de definiție de tip de structură:

```
ALFA STRUC
      a db ?
      b dw 1111H
      c dd 1234
      d db ?
ALFA ENDS
```

O structură definită este interpretată ca un tip nou de date, care poate apoi participa la definiții concrete de date, cu rezervare de spațiu de memorie. De exemplu, putem defini o structură de tipul ALFA, cu numele x:

```
x ALFA <, , 777, 5>
```

în care ALFA este tipul de date, x este numele variabilei, iar între paranteze unghiulare se trec valorile inițiale ale câmpurilor structurii x. Prezența virgulelor din lista de valori inițiale precizează că primii doi membri sunt inițializați cu valorile implicite de la definiția tipului ALFA, al treilea membru este inițializat cu valoarea 777, iar al patrulea cu valoarea 5. Astfel, definiția variabilei x este echivalentă cu secvența de definiții:

```
a db ?
b dw 1111H
c dd 777
d db 5
```

Principalul avantaj al structurilor este accesul la membri într-o formă asemănătoare limbajelor de nivel înalt. De exemplu, pentru a încărca în SI câmpul b al structurii x, putem scrie:

```
mov si, x.b
```

Accesul la membrii structurii x poate fi deci detaliat în:

- x.a - variabilă de tip byte
- x.b - variabilă de tip word
- x.c - variabilă de tip dword
- x.d - variabilă de tip byte

Putem acum defini un tablou de 5 structuri de tip ALFA:

```
y ALFA 5 dup <, , , 10>
```

în care primii trei membri sunt inițializați cu valorile de la definiția tipului structurii, iar al patrulea cu valoarea 10. Au acum sens expresii de forma: y[0].a, y[SI], y[8].c, y[BX].d etc. De fapt, toate aceste expresii se evaluează din adresa de început a variabilei y la care se adaugă un deplasament corespunzător. Expresia y[11].d este echivalentă cu y + 11 + deplasamentul câmpului d față în cadrul tipului ALFA.

Se observă că gestiunea deplasamentelor cade în sarcina utilizatorului. Dacă dorim să accesăm câmpul c al celei de-a doua structuri din tabloul y, nu putem

scrie y[1].c, așa cum am scrie într-un limbaj de nivel înalt, ci y[8].c. Limbajul de asamblare ne pune la dispoziție operatorul TYPE, care întoarce numărul de octeți ocupat de o structură. Pentru a generaliza accesul la al i-lea element al tabloului y (i cuprins între 0 și 4), putem scrie y[i*TYPE ALFA].c.

Problema se complică dacă indicele i este cunoscut de-abia la momentul execuției (fiind memorat într-un registru, de exemplu). O expresie de forma:

```
mov al, y [SI*TYPE ALFA].d
```

are sens numai la procesoarele care acceptă adresare cu factor de scală (80386 și peste) și numai dacă factorul de scală este 1, 2, 4, sau 8. În caz contrar, deplasamentul trebuie calculat explicit prin operații aritmetice.

Cu toate aceste limitări, structurile sunt intens utilizate atunci când se lucrează cu anumite șabloane fixe, cum ar fi, de exemplu, imaginea stivei la intrarea într-o procedură sau transmiterea prin referință a unei zone de date. Să considerăm o înregistrare logică descrisă prin câmpuri:

- câmpul NUME - șir ASCII de 30 de caractere
- câmpul PRENUME - șir ASCII de 20 de caractere
- câmpul COD - întreg pe 4 octeți

Dorim să transmitem unei proceduri far cu numele PRELUCRARE o asemenea înregistrare. Cel mai simplu mod este de a transmite adresa (near sau far) a zonei de memorie în care este stocată înregistrarea, de exemplu printr-un registru sau o pereche de registre; pentru accesul comod la câmpuri definim structura:

```
PERSOANA STRUC
          NUME        db 30 dup (?)
          PRENUME     db 20 dup (?)
          COD         dd ?
PERSOANA ENDS
```

Definiția concretă a datelor și secvența de apel a procedurii este:

```
.data
      om          PERSOANA <'Ionescu', 'Ion', 77206>
      adr_om      dd om                ; Pointer la om
.code
      les di, adr_om
      call PRELUCRARE
```

Se pune acum problema accesului la câmpurile înregistrării, din interiorul procedurii. Beneficiem acum de structura PERSOANA. Ca să încărcăm adresa câmpului PRENUME vom scrie:

```
lea bx, es:[di].PRENUME
```

Perechea ES:DI conține adresa structurii om, iar expresia es:[di].PRENUME reprezintă deplasamentul câmpului PRENUME. În acest moment, în ES:BX avem adresa șirului de caractere de tip PRENUME. Similar se procedează și pentru celelalte câmpuri: ca să preluăm în DX:AX câmpul COD, putem scrie:

```
mov ax, word ptr es:[di].COD
mov dx, word ptr es:[di].COD + 2
```

Este evident că toate secvențele de mai sus se puteau înlocui cu deplasamente calculate explicit de către programator. Încărcarea câmpului COD ar fi putut fi scrisă:

```
mov ax, word ptr es:[di + 50]
mov dx, word ptr es:[di + 52]
```

dar forma cu nume de câmpuri este mult mai clară și mai comodă. În plus, noi putem greși la calculul unui deplasament, dar asamblorul nu.

Această tehnică este benefică și din punctul de vedere al întreținerii unui program. Dacă se modifică ulterior înregistrarea PERSOANA și se adaugă încă un câmp între PRENUME și COD, toate deplasamentele calculate explicit trebuie modificate. În varianta cu structură, trebuie doar să schimbăm definiția structurii, restul fiind făcut automat la asamblare.

4.8 Definirea înregistrărilor. Operații specifice

Înregistrările (RECORD) corespund de fapt unor structuri împachetate din limbajele de nivel înalt. Concret, o înregistrare este o definiție de câmpuri de biți de lungime maximă 8 sau 16. Din punct de vedere sintactic, definiția unei înregistrări este similară cu cea a unei structuri, forma generală fiind:

```
nume_inregistrare RECORD nume_camp:valoare, ...
```

sau, cu inițializare implicită:

```
nume RECORD nume_camp:valoare = expresie, ...
```

Valorile care apar asociate cu numele de câmpuri dau de fapt numărul de biți pe care se memorează câmpul respectiv. În varianta cu inițializare, expresia care apare după semnul egal se evaluează la o cantitate care se reprezintă pe numărul de biți asociat câmpului respectiv. La fel ca la structuri, numele câmpurilor trebuie să fie distincte, chiar dacă aparțin unor înregistrări diferite.

Să considerăm un exemplu:

BETA record X:7, Y:4, Z:5

prin care se definește un șablon de 16 biți, grupați în câmpurile X, Y și Z (vezi Figura 4.2)

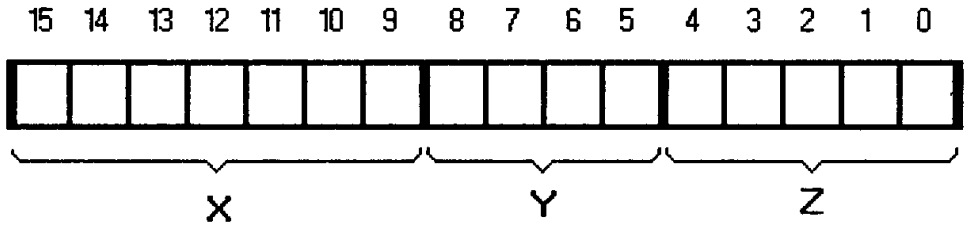


Figura 4.2 Definiția unei înregistrări

La fel ca la structuri, putem acum defini variabile de tipul înregistrării BETA:

```
VAL BETA <5, 2, 7>
```

în care valorile dintre parantezele unghiulare inițializează cele trei câmpuri de biți. Se observă că această definiție este echivalentă cu definiția explicită:

```
VAL dw 0000101001000111B
```

Există doi operatori specifici înregistrărilor: MASK și WIDTH. Operatorul MASK primește un nume de câmp și furnizează o mască cu biții 1 pe poziția câmpului respectiv și 0 în rest. Astfel, expresiile MASK X și MASK Y sunt echivalente cu constantele binare 1111111000000000B și 0000000111100000B.

Asemenea expresii pot fi utilizate pentru selectarea câmpurilor respective, ca în exemplele de mai jos:

```
AND AX, MASK Y          ; Filtreaza campul Y
AND BX, NOT MASK Z      ; Forteaza campul Z la 0
```

Operatorul WIDTH primește un nume de înregistrare sau un nume de câmp dintr-o înregistrare, întorcând numărul de biți ai înregistrării sau ai câmpului respectiv. De exemplu, secvența:

```
MOV AL, WIDTH BETA      ; Al <--- 16
MOV BL, WIDTH Y         ; BL <--- 4
```

va încărca în AL valoarea 16 și în BL valoarea 4.

Dacă se utilizează un nume de câmp într-o instrucțiune de tip MOV, se va încărca un contor de deplasare, util pentru a deplasa câmpul respectiv pe pozițiile cele mai puțin semnificative.

Să presupunem că dorim să testăm valoarea numerică a câmpului Y din înregistrarea VAL. Nu este suficient să-l izolăm și să folosim o instrucțiune de comparație. Înainte de comparație, câmpul Y trebuie adus pe pozițiile cele mai din dreapta. Realizăm aceste operații prin secvența:

```
MOV AX, VAL        ; Preluare inregistrare
AND AX, MASK Y     ; Selectare camp Y
MOV CL, WIDTH Y    ; Incarcare in CL a valorii 4
SHR AX, CL         ; Deplasare camp Y la dreapta
```

4.9 Operatori în limbajul de asamblare

Limbajul de asamblare dispune de un set de operatori cu care se pot forma expresii aritmetice și logice. Aceste expresii sunt evaluate la momentul asamblării, producând, de fapt, constante numerice. Este esențial să deosebim instrucțiunile executabile (codul mașină) de operațiile care se fac la momentul asamblării.

4.9.1 Operatori aritmetici și logici

Operatorii aritmetici sunt: +, -, *, /, MOD, SHL și SHR. Primii patru au semnificații evidente și operează numai cu cantități întregi. Operatorul MOD produce restul la împărțire, iar SHL și SHR provoacă deplasare la stânga sau la dreapta.

De exemplu, instrucțiunea:

```
mov ax, 1 SHL 3          ; 1 deplasat la stanga cu 3 biti
```

este echivalentă cu:

```
mov ax, 100B
```

Similar, în directiva:

```
COUNT dw ($ - TAB)/2
```

se va evalua la momentul asamblării expresia ($- TAB)/2, adică diferența dintre contorul curent de locații și adresa TAB, împărțită la 2.

Operatorii logici sunt NOT, AND, OR și XOR, cu semnificații evidente. Toți acești operatori acceptă drept operanzi constante întregi; operațiile respective se fac la nivel de bit. Ei nu trebuie confundați cu instrucțiunile executabile cu același nume. În instrucțiunea:

```
and al, (1 SHL 3 ) OR (1 SHL 5)
```

registrul AL va fi încărcat cu constanta 101000B.

4.9.2 Operatori relaționali

Operatorii relaționali sunt EQ, NE, LT, LE, GT, GE, cu semnificații evidente. Aceștia întorc valori logice, codificate ca 0 sau ca secvențe de 1 pe un număr corespunzător de biți. În urma definițiilor:

```
x db 1 EQ 2
y dw 1 NE 2
z dd 1 LT 2
```

variabilele x, y și z vor fi inițializate cu constantele 0, 0FFFFH, respectiv 0FFFFFFFFH.

Operatorii relaționali sunt utilizați în special în directivele de asamblare condiționată.

4.9.3 Operatorul de atribuire =

Operatorul de atribuire = definește constante simbolice, fiind similar cu directiva EQU, dar permite redefinirea simbolurilor utilizate. O definire de forma:

```
N EQU 1
N EQU 2
```

este semnalată ca eroare, dar secvența:

```
N = 1
N = 2
```

este corectă.

Acest operator este utilizat în special în definițiile de macroinstrucțiuni.

4.9.4 Operatori care întorc valori

Acești operatori se aplică unor entități ale programului în limbaj de asamblare (variabile, etichete), întorcând valori asociate acestora.

- **Operatorul SEG**

Se aplică atât variabilelor, cât și etichetelor și furnizează adresa de segment asociată variabilei respective. Dacă var este o variabilă, atunci putem scrie secvența:

```
mov ax, SEG var
mov ds, ax
```

- **Operatorul OFFSET**

Este similar cu SEG, furnizând offset-ul asociat variabilei sau etichetei:

```
mov bx, OFFSET var
```

- **Operatorul THIS**

Acest operator creează un operand care are asociate o adresă de segment și un offset identice cu contorul curent de locații. Forma generală a operandului este:

```
THIS tip
```

în care tip poate fi BYTE, WORD, DWORD, QWORD sau TBYTE pentru definiții de date, respectiv NEAR sau FAR pentru etichete. Operatorul THIS se

utilizează de obicei cu directiva EQU. De exemplu, definiția constantei simbolice:

```
ALFA EQU THIS WORD
```

este echivalentă cu definiția unei etichete:

```
ALFA LABEL WORD
```

- **Operatorul TYPE**

Se aplică variabilelor și etichetelor, întorcând tipul acestora. Pentru variabile întoarce valorile 1, 2, 4, 8 sau 10 pentru variabile simple (definite cu directivele DB, DW, DD, DQ respectiv DT), iar pentru structuri întoarce numărul de octeți pe care este memorată structura respectivă. Pentru etichete, întoarce tipul etichetei (NEAR sau FAR).

- **Operatorul LENGTH**

Se aplică numai variabilelor și întoarce numărul de elemente definite în variabila respectivă. De exemplu, în definiția:

```
A DW 100 dup (?)
```

expresia LENGTH A are valoarea 100.

- **Operatorul SIZE**

Se aplică numai variabilelor și întoarce dimensiunea în octeți a variabilei respective. În definiția de mai sus, expresia SIZE A are valoarea 200.

Pentru o variabilă oarecare var, are loc totdeauna identitatea:

```
SIZE var = (LENGTH var) * (TYPE var)
```

Acești operatori sunt utili la prelucrarea tablourilor. Secvența următoare nu depinde de tipul de bază al tabloului TAB și nici de dimensiunea sa:

```
.data
      TAB   dd 100 dup (?)
.code
      mov   cx, LENGTH TAB
      lea   si, TAB
bucla:
      ....
      add   si, TYPE TAB
      loop  bucla
```

Dacă înlocuim acum tipul de bază al tabloului TAB cu o structură de tip PERSOANA (definită în 4.7), modificând și dimensiunea:

```
PERSOANA STRUC
      .......
PERSOANA ENDS
.data
```

```
TAB    PERSOANA 50 dup <,,,>
```

partea de program nu trebuie modificată, deoarece asamblorul va evalua corect expresiile LENGTH TAB și TYPE TAB.

4.9.5 Operatorul PTR

Acest operator se aplică atât variabilelor, cât și etichetelor, având ca efect schimbarea tipului variabilei sau al etichetei respective. Este obligatoriu în cazul în care se folosesc referințe anonime la memorie, din care nu se poate deduce tipul operandului:

```
add   word ptr [bx], 2
call  dword ptr [bx]
call  near ptr proc
```

4.10 Directive de asamblare condiționată

Permit ignorarea unor porțiuni din textul sursă, funcție de o condiție care se poate evalua la asamblare. Directiva de bază este IF, cu forma generală:

```
IF expresie
        .
        .
        .
[ELSE]
        .
        .
        .
ENDIF
```

Dacă expresia din directiva IF este adevărată (adică este diferită de 0), se asamblează porțiunea de program dintre IF și ELSE și se ignoră porțiunea dintre ELSE și ENDIF. Dacă expresia din IF este falsă, se asamblează porțiunea de program dintre ELSE și ENDIF și se ignoră porțiunea dintre IF și ELSE. Ramura ELSE este opțională.

Aceste operații se petrec la momentul asamblării, permițând întreținerea comodă a unui program de dimensiuni mari sau chiar menținerea în același text sursă a mai multor variante de program executabil. Să presupunem că dorim un program care să poată fi configurat rapid pentru modele de date „mici", respectiv „mari". Aceasta presupune ca orice secvență de instrucțiuni care definește, citește sau scrie o adresă să fie inclusă în directive de asamblare condiționată:

```
FALSE        equ 0
TRUE         equ NOT FALSE
DATE_MARI    equ TRUE
DATE_MICI    equ NOT DATE_MARI
.data
      X dw 100 dup (?)
IF DATE_MARI
```

```
        ADR_X       dd X
ELSE
        ADR_X       dw X
ENDIF
.code
IF DATE_MARI
        lds   si, ADR_X
ELSE
        mov   si, ADR_X
ENDIF
;
; Prelucrare TABLOU
;
```

Există și directivele IFDEF/ENDIF și IFNDEF/ENDIF, care testează dacă un simbol este definit sau nu, și directivele IFB/ENDIF (If Blank), respectiv IFNB/ENDIF (If Not Blank), care testează dacă un simbol este vid sau nevid. Acestea din urmă vor fi utilizate la definirea macroinstrucțiunilor.

În expresiile care apar în directivele de tip IF se utilizează de obicei operatori logici și relaționali.

Să considerăm un exemplu de program sursă multifuncțional. Se dorește scrierea unei secvențe care să calculeze în acumulator suma elementelor unui tablou A (de octeți sau de cuvinte), definit în segmentul curent de date, ca sumă pe octeți și pe cuvinte, indiferent de tipul și de dimensiunea tabloului. Tipul sumei este dictat de constantele simbolice SUM_B și SUM_W.

```
        FALSE       equ 0
        TRUE        equ NOT FALSE
        SUM_B       equ TRUE
        SUM_W       equ NOT SUM_B
        .code
if SUM_B
        mov   cx, SIZE A                ; Numar de iteratii
else
        mov   cx, (SIZE A)/2
endif
        xor   bx, bx                    ; Indice initial
        mov   ax, bx                    ; Suma initiala
bucla:
if SUM_B
        add   al, byte ptr A [bx]       ; Suma pe octet
        inc   bx                        ; Actualizare adresa
else
        add   ax, word ptr A [bx]       ; Suma pe cuvant
        inc   bx                        ; Actualizare adresa
        inc bx
endif
        loop bucla                      ; Bucla

if SUM_W AND (TYPE A EQ 1) AND ((SIZE A) MOD 2) EQ 1
        push  ax
```

```
        mov     al, byte ptr A [bx]      ; Un eventual
        cbw                              ; ultim
        mov     dx, ax                   ; octet
        pop     ax
        add     ax, dx
endif
```

Se observă că nu putem inițializa contorul CX cu expresia LENGTH A, deoarece numărul de iterații este dictat de modul de calcul al sumei, și nu de tipul tabloului. Ca atare, luăm dimensiunea în octeți a tabloului, pe care o împărțim la 2 în cazul sumei pe cuvânt. Calculul sumei și actualizarea adresei sunt evidente.

Apare o problemă specială în următoarea situație: se cere suma pe cuvânt, iar tabloul, definit ca tablou de octeți, are un număr impar de elemente. În această situație, este evident că ultimul octet nu este considerat în bucla de sumare. Ca atare, el este adăugat explicit la sfârșit.

De remarcat formularea situației de mai sus în termenii condiției logice de la ultima directivă IF:

- SUM_W codifică cererea de calcul a sumei la nivel de cuvânt;
- (TYPE A EQ 1) codifică situația în care tabloul A este definit la nivel de octet;
- ((SIZE A) MOD 2) codifică situația în care tabloul are un număr impar de octeți.

Capitolul 5

Macroinstrucțiuni

5.1 Scopul macroinstrucțiunilor. Definire și expandare

Macroinstrucțiunile permit programatorului să definească simbolic secvențe de program (instrucțiuni, definiții de date, directive etc.), asociate cu un nume. Folosind numele macroinstrucțiunii în program, se va genera întreaga secvență de program. În esență, este vorba de un proces de substituție (expandare) în textul programului sursă, care se petrece înainte de asamblarea programului. Un asamblor care dispune de macroinstrucțiuni se numește macroasamblor.

Sunt două etape de lucru cu macroinstrucțiuni: definirea macroinstrucțiunilor și utilizarea lor. Utilizarea se mai numește invocare sau chiar apel de macroinstrucțiune, dar ultima denumire poate produce confuzie, termenul utilizându-se în cazul procedurilor.

Spre deosebire de proceduri, macroinstrucțiunile sunt expandate la fiecare utilizare, deci programul nu se micșorează. Avantajul este că textul sursă scris de programator devine mai clar și mai scurt.

Macroinstrucțiunile pot fi cu parametri sau fără parametri. Din punct de vedere sintactic, ele sunt asemănătoare cu directivele de tip DEFINE din limbajele de nivel înalt, dispunând, în general, de mecanisme mai evoluate decât acestea.

Definiția unei macroinstrucțiuni fără parametri se face în forma generală:

```
nume_macro MACRO
        ;
        ; Corp macroinstructiune
        ;
ENDM
```

Invocarea constă în scrierea în textul sursă a numelui macroinstrucțiunii. Macroinstrucțiunea init_ds_es este definită în fișierul header io.h prin:

```
init_ds_es macro
        mov    ax, DGROUP
        mov    ds, ax
        mov    es, ax
endm
```

Macroinstrucțiunea exit_dos este definită prin:

```
exit_dos macro
        mov ax, 4C00H
       int 21H
endm
```

O pereche de macroinstrucțiuni care salvează și refac registrele generale poate fi concepută astfel:

```
save   macro
       push  ax
       push  bx
       push  cx
       push  dx
       push  si
       push  di
 endm
```

```
rest   macro
       pop   di
       pop   si
       pop   dx
       pop   cx
       pop   bx
       pop   ax
endm
```

Putem utiliza aceste macroinstrucțiuni la intrarea și la ieșirea dintr-o procedură:

```
PROCEDURA    proc near
       save
       .
       .
       rest
       ret
PROCEDURA    endp
```

Din punct de vedere simbolic, este ca și cum setul de instrucțiuni al mașinii ar fi fost extins cu două noi instrucțiuni: SAVE și REST.

5.2 Macroinstrucțiuni cu parametri

Macroinstrucțiunile importante sunt cele cu parametri. Definiția unei macroinstrucțiuni cu parametri are forma generală:

```
nume_macro  MACRO P1, P2, ..., Pn
       ;
       ; Corp macroinstructiune
       ;
ENDM
```

în care P1, P2, ..., Pn sunt identificatori care specifică parametrii formali. Apelul (invocarea) unei macroinstrucțiuni cu parametri se face prin specificarea numelui, urmată de o listă de parametri actuali:

```
nume_macro  X1, X2, ..., Xn
```

La expandarea macroinstrucțiunii, pe lângă expandarea propriu-zisă, se va înlocui fiecare parametru formal cu parametrul actual respectiv.

Să considerăm câteva exemple. Apelurile de funcții DOS presupun numărul funcției în registrul AH. Putem, deci, defini o macroinstrucțiune de forma:

```
dosint      macro N
            mov   ah, N
            int   21H
endm
```

și putem invoca macroinstrucțiunea prin linii de forma:

```
dosint      2
```

Similar, pentru deschiderea unui fișier disc pentru citire (operație realizată prin funcția DOS 3CH), putem scrie o macroinstrucțiune de forma:

```
o_read macro      fname, hand
       mov        al, 0C0H
       lea        dx, fname
       dosint     3DH
       mov        hand, ax
endm
```

în care file_name conține numele fișierului, iar handle este o variabilă de tip word în care se depune un indicator către fișierul deschis. Parametrul 0C0H codifică modul de acces. Se observă utilizarea macroinstrucțiunii dos_int în definiția lui o_read.

Citirea din fișier poate fi codificată într-o macroinstrucțiune de forma:

```
f_read macro hand, buf, nr
       mov        bx, hand
       mov        cx, nr
       lea        dx, buf
       dosint     3FH
endm
```

în care hand este indicatorul către fișierul anterior deschis, nr este numărul de octeți care se citește, iar buf este adresa unei zone de memorie în care se vor depune datele citite.

Închiderea unui fișier deschis se poate face cu o macroinstrucțiune de forma:

```
f_close     macro hand
       mov        bx, hand
       dosint     3EH
endm
```

Macroinstrucțiunile de mai sus sunt definite în fișierul io.h. Ne propunem acum să scriem un program executabil care să afișeze la consolă un fișier text. Beneficiem de macroinstrucțiunile definite în fișierul io.h:

```
.model      large
include     io.h
.stack      1024
.data
       file_name   db    30 dup (0)   ; Spatiu nume fisier
       hand        dw    ?            ; Spatiu handler
```

```
            buf         db    1024 dup (?); Buffer citire
      .code
start:
      init_ds_es
      putsi       <'Nume fisier: '>       ; Mesaj
      gets    file_name                   ; Citire nume
      o_read file_name, hand              ; Deschidere
      jc      eroare                      ; Eroare deschidere ?
bucla:
      f_read      hand, buf, 1024         ; Citire 1024 octeti
      jc      eroare                      ; Eroare citire
      mov     si, ax                      ; AX = cati octeti s-au
      mov     byte ptr buf [si], 0        ; citit de fapt
      puts    buf                         ; Afisare la consola
      cmp     ax, 1024                    ; S-au citit mai putin
      jb      gata                        ; de 1024 ?
      jmp     bucla                       ; Nu, reluare
gata:
      f_close     hand                    ; Inchidere
      jmp   iesire                        ; Salt la iesire
eroare:
                                          ; Mesaj de
      putsi       <'Eroare fisier'>       ; eroare
iesire:
      exit_dos                            ; Iesire în DOS
end   start
```

În zona de date, se rezervă spațiu pentru numele fișierului, pentru indicator (handler) și pentru un buffer de citire de 1024 de octeți. Reamintim că operațiile cu perifericele sunt în esență transferuri între periferice și memorie.

Programul afișează un mesaj la consolă, după care citește un nume de fișier. Se face apoi operația de deschidere a fișierului specificat. Toate funcțiile DOS de lucru cu fișiere întorc CF = 1 în caz de eroare. O eroare tipică la operația de deschidere pentru citire este un nume eronat de fișier. Se testează deci CF și, în caz de eroare, se afișează un mesaj adecvat și se iese în DOS.

Se trece acum la o buclă de citire-afișare. Funcția de citire întoarce în AX numărul de octeți efectiv citiți (care este mai mic sau egal cu cel cerut). Dorim să afișăm numai ce s-a citit efectiv, așa că punem terminatorul 0 în buffer, după ultimul octet efectiv citit. E posibil ca, la ultima iterație, să se citească 0 octeți. Se afișează la consolă bufferul respectiv (cu macroinstrucțiunea puts).

Dacă numărul de octeți efectiv citiți este mai mic strict decât cel cerut (1024) înseamnă că s-a ajuns la sfârșitul fișierului și bucla se termină. În caz contrar, se reia cu o nouă citire din fișier. În final, se închide fișierul și se iese în DOS.

Acest exemplu ilustrează foarte bine avantajele macroinstrucțiunilor. O acțiune destul de laborioasă în limbaj de asamblare (afișare fișiere text) a putut fi codificată prin câteva linii de program sursă (e drept că aproape toate sunt invocări de macroinstrucțiuni).

Morala fabulei este că, dacă reușim să concepem un set de macroinstrucțiuni adecvat unei probleme (în cazul de față, interfața cu sistemul DOS), scrierea programelor devine foarte comodă.

Cititorul este sfătuit să scrie un program similar, care să realizeze copierea unui fișier disc în alt fișier. Pentru operații de scriere, se pot utiliza macroinstrucțiunile:

```
o_write     macro fname, hand
      mov         al, 0C1H
      lea         dx, fname
      dosint      3DH
      move        hand, ax
endm
f_write     macro hand, buf, nr
      mov         bx, hand
      mov         cx, nr
      lea         dx, buf
      dosint      40H
endm
```

care deschid un fișier pentru scriere, respectiv scriu în fișier. Parametrii sunt asemănători cu cei din macroinstrucțiunile o_read și f_read. Bucla de citire_din_fișier-afișare din exemplul anterior se înlocuiește cu o buclă de citire_din_fișier_sursă- scriere_în_fișier_destinație.

În unele situații, substituția parametrilor formali cu cei actuali poate ridica unele probleme. Să presupunem că un tânăr programator care învață limbajul ASM nu a ajuns încă la instrucțiunea XCHG, ci are cunoștință doar de instrucțiunile PUSH, POP și MOV. El își propune să scrie o macroinstrucțiune care să interschimbe două cantități de 16 biți:

```
trans macro X, Y
      push  ax
      push  bx
      mov   bx, X
      mov   ax, Y
      mov   X, ax
      mov   Y, bx
      pop   bx
      pop   ax
endm
```

Aparent, totul e în ordine. Totuși, pot apărea situații nedorite, ca în secvența:

```
trans       AX, SI      ; Interschimba AX cu SI (oare ?)
```

Această invocare de macroinstrucțiune se expandează în:

```
push  ax
push  bx
mov   bx, AX
mov   ax, SI
```

```
mov   AX, ax
mov   SI, bx
pop   bx
pop   ax
```

și este evident că registrul AX nu se modifică. Se poate însă și mai rău, ca în secvența:

```
trans       SP, DI      ; Interschimba SP cu DI (oare ?)
```

care se expandează în:

```
push  ax
push  bx
mov   bx, SP
mov   ax, DI
mov   SP, ax            ; Aici se modifica SP
mov   DI, bx            ; si POP-urile
pop   bx                ; sunt
pop   ax                ; compromise
```

Pericolul apare, deci, în situațiile în care parametrii actuali intră în conflict cu anumite variabile sau registre care sunt folosite în interiorul macroinstrucțiunii. Aceste situații trebuie evident evitate.

Utilitatea foarte mare a macroinstrucțiunilor devine evidentă la secvențele de apel ale procedurilor cu parametri (în acest caz, se vorbește despre macroinstrucțiuni de apel). Apelurile de funcții sistem codificate anterior sunt cazuri particulare de macroinstrucțiuni de apel.

Procedurile cu parametri utilizează diverse tehnici de transmitere a parametrilor. Parametrii trebuie plasați în anumite registre sau în stivă, într-o ordine specificată. Aceste detalii de apel sunt greu de ținut minte și nici nu interesează pe cel care apelează procedura. Putem însă dezvolta macroinstrucțiuni care să ascundă aceste detalii.

Să considerăm o procedură NEAR cu numele puts_proc, care afișează un șir de caractere (terminat cu 0). Adresa șirului (offset-ul în cadrul segmentului curent adresat prin DS) se specifică în registrul SI. O macroinstrucțiune de apel se poate scrie în forma:

```
puts  macro X
      push  si
      lea   si, X
      call  puts_proc
      pop   si
endm
```

Ca parametru actual se poate utiliza orice operand compatibil cu instrucțiunea LEA. Astfel, dacă există definiția de date:

```
.data
```

```
STRING      db    "abcdefghijklmnopqrstuvwxyz", 0
```

se pot utiliza formele de invocare:

```
.code
      puts  STRING       ; Adresare directa
      lea   bx, STRING
      puts  [bx]         ; Adresare indirecta
```

În al doilea caz, nu putem scrie puts bx, pentru că acest apel ar conduce la o expandare de forma lea si, bx, ceea ce este o eroare de sintaxă. Forma puts [bx] se expandează în lea si, [bx], care este corectă.

5.3 Controlul numărului de parametri actuali

O problemă importantă este controlul coincidenței dintre numărul parametrilor formali și cel al parametrilor actuali. Se vor utiliza următoarele directive specifice:

- Directiva %OUT (Mesaj la consolă la momentul asamblării). Are forma generală %OUT TEXT, efectul fiind afișarea TEXT-ului la consolă, la momentul asamblării; este utilizată pentru mesaje de eroare la asamblare.
- Directiva .ERR (Forțează eroare de asamblare). Această directivă forțează o eroare la asamblare, astfel încât să nu se mai genereze fișier obiect; este utilizată în cazul neconcordanței dintre numărul parametrilor actuali și cel al parametrilor formali.
- Directiva EXITM (Ieșire forțată din expandare)

În caz de eroare, stopăm expandarea macroinstrucțiunii curente și afișăm un mesaj de eroare la consolă.

Pentru controlul efectiv al parametrilor actuali, se mai utilizează directivele de asamblare condiționată IFNB (If Not Blank) și IFB (If Blank) care testează existența sau non- existența unui parametru actual.

Pentru forțarea unei erori, definim pentru început macroinstrucțiunea:

```
eroare      macro text
            %OUT  text          ;; Mesaj la consola
            .err                ;; Fortare eroare la
                                ;; asamblare
endm
```

Să considerăm macroinstrucțiunea puts, definită mai sus, care are un singur parametru formal. La invocare, s-ar putea să existe un parametru actual, nici unul sau mai mulți. Pentru a controla toate aceste situații, rescriem macroinstrucțiunea puts și o declarăm cu doi parametri: X și IN_PLUS. Dacă IN_PLUS este nevid, înseamnă că sunt cel puțin doi parametri actuali, iar dacă X este vid, nu există nici un parametru actual. Codificarea va fi deci:

```
puts  macro X, IN_PLUS
```

```
        ifb     <X>
                eroare      <PUTS - lipsa parametru>
                exitm
        endif
        ifnb    <IN_PLUS>
                eroare      <PUTS - parametri in plus>
                exitm
        endif
        lea     si, X
        call    puts_proc
endm
```

Testăm explicit dacă X este vid și dacă IN_PLUS este nevid, forțând mesaje adecvate de eroare. Scrierea cu paranteze unghiulare <> înseamnă literalizarea textului respectiv, adică transmiterea lui ca un unic parametru. Dacă se consideră un fișier t_macro.asm cu secvențele de invocare:

```
.data
        sir     db 10 dup (0)
.code
        puts    [si]
        puts    [bx] [si]
        puts    [bx], [si]
        puts
        puts    [bx], sir
        puts    sir
end
```

atunci se vor obține mesajele de eroare:

```
PUTS - parametri in plus
PUTS - parametri in plus
PUTS - lipsa parametru
PUTS - parametri in plus
**Error** t_macro.asm(26) EROARE(2) User generated error
**Error** t_macro.asm(27) EROARE(2) User generated error
**Error** t_macro.asm(28) EROARE(2) User generated error
**Error** t_macro.asm(29) EROARE(2) User generated error
```

Numerele din paranteze indică liniile din fișierul sursă în care a fost forțată explicit eroarea de asamblare.

Putem acum da o regulă generală de scriere a unei macroinstrucțiuni cu controlul parametrilor. Dacă macroinstrucțiunea are N parametri utili, declarăm un al N+1-lea parametru suplimentar și testăm dacă al N-lea parametru este vid (cu directiva IFB), respectiv dacă al N+1-lea parametru este nevid (cu directiva IFNB).

5.4 Directive de control al listării. Etichete în macroinstrucțiuni

Există diverse posibilități de control al fișierului listing al unui program care conține macroinstrucțiuni. Controlul se face prin următoarele directive:

- Directiva .SALL (Supress All - Suprimă tot). În textul sursă care urmează acestei directive, se va suprima listarea conținutului macroinstrucțiunilor invocate. În listing apare numai invocarea (numele) macroinstrucțiunii, exact ca în fișierul sursă.
- Directiva .LALL (List All - Listează tot). După această directivă, se listează atât invocarea macroinstrucțiunii, cât și textul generat.
- Directiva .XALL - După această directivă, se listează textul generat efectiv de invocarea unei macroinstrucțiuni. Dacă macroinstrucțiunile conțin invocări ale altor macroinstrucțiuni, la al doilea nivel se listează numai invocarea.

Uneori e necesar ca macroinstrucțiunile să conțină etichete, de exemplu pentru instrucțiuni de salt. Apare însă următoarea problemă: dacă eticheta se definește în mod obișnuit, ea va fi generată la fiecare expandare a macroinstrucțiunii, ceea ce va duce la mai multe etichete cu același nume. Acest fapt va provoca o eroare la asamblare, de tip „simbol multiplu definit".

Problema se rezolvă prin directiva LOCAL, care are forma generală:

```
LOCAL        simb_1, simb_2, ...
```

efectul fiind că simbolurile care apar în directivă vor fi expandate în așa fel încât să nu apară două nume identice. Practic, asamblorul generează niște nume complicate, de forma ??0000, ??0001, ??0002 etc. (care nu se repetă), pentru care există șanse minime să coincidă cu nume definite de utilizator.

Să considerăm macroinstrucțiunea:

```
test_local macro
local aici
      jmp   aici
aici:
endm
```

și secvența de invocare:

```
.code
.xall
      test_local
      test_local
      test_local
end
```

Se va genera următorul listing:

```
test_local
        jmp     ??0000
??0000:
test_local
        jmp     ??0001
??0001:
test_local
        jmp     ??0002
??0002:
```

Eticheta aici a fost înlocuită la expandare cu etichetele ??0000, ??0001 și ??0002.

Simbolurile variabile (definite cu operatorul de atribuire =) se dovedesc utile în macroinstrucțiuni, deoarece pot fi redefinite chiar în macroinstrucțiune. Să considerăm definiția:

```
m_mesaj macro
        db      'Mesaj ', '0'+n, 0
        n = n + 1
endm
```

și secvența de program:

```
n = 1
.data
        m_mesaj
        m_mesaj
        m_mesaj
```

Se va genera textul sursă:

```
db      'Mesaj ', '0' + 1, 0
db      'Mesaj ', '0' + 2, 0
db      'Mesaj ', '0' + 3, 0
```

deci ca și cum s-ar fi scris:

```
db      'Mesaj 1', 0
db      'Mesaj 2', 0
db      'Mesaj 3', 0
```

Dacă dorim să includem comentarii în macroinstrucțiuni, dar acestea să nu apară la fiecare expandare, ci numai la definiția macroinstrucțiunii, putem utiliza secvența de două caractere ;; (început de comentariu).

5.5 Macroinstrucțiuni repetitive

Aceste macroinstrucțiuni sunt predefinite, deci nu trebuie definite de utilizator. Scopul lor este de a genera secvențe repetate de program.

- **Macroinstrucțiunea REPT (Repete - Repetă)**

Forma generală de invocare este:

```
rept  n
      ;
      ; Corp macroinstructiune
      ;
endm
```

în care n este o constantă întreagă. Efectul este repetarea corpului macroinstrucțiunii de n ori. Secvența anterioară de generare a trei mesaje s-ar putea scrie acum:

```
rept 3
      m_mesaj
endm
```

Iată o secvență care generează un șir de caractere cu litere de la 'A' la 'Z':

```
n = 0
alfabet label byte
rept 26
db 'A'+n
n = n + 1
endm
```

- **Macroinstrucțiunea IRP (Indefinite Repeate - Repetă nedefinit)**

Forma generală de invocare este:

```
irp p_formal, <lista_par_act>
      ;
      ; Corp macroinstructiune
      ;
endm
```

în care p_formal este un parametru formal, iar listă_par_act este o listă de parametri actuali, separați prin virgulă. Efectul este repetarea corpului macroinstrucțiunii de atâtea ori câte elemente conține lista de parametri actuali. La fiecare repetare, se substituie parametrul formal cu câte un parametru actual. Invocarea:

```
irp x, <'a', 'b', 'c'>
 db x
endm
```

se va expanda în:

```
db 'a'
db 'b'
db 'c'
```

5.6 Operatori specifici

Există o serie de operatori specifici macroinstrucțiunilor. Aceștia controlează în principal substituția parametrilor actuali, fiind utili în special în macroinstrucțiunile repetitive.

5.6.1 Operatorul de substituire și de concatenare (&)

Acest operator, aplicat unui parametru formal, realizează substituția și (eventual) concatenarea sa cu un text fix sau cu un alt parametru formal. Este necesar în contextul în care un parametru formal ar fi interpretat ca un simbol (de exemplu, într-un șir constant de caractere sau într-un identificator).

Să presupunem că dorim să definim automat liniile de program:

```
mesaj_1 db 'Text 1', 0
mesaj_2 db 'Text 2', 0
mesaj_3 db 'Text 3', 0
mesaj_4 db 'Text 4', 0
```

Vom apela evident la macroinstrucțiunea IRP. O primă încercare ar fi:

```
irp x, < 1, 2, 3, 4 >
       mesaj_x db 'Text x', 0
endm
```

ceea ce este incorect, deoarece se va produce aceeași linie de program. Prima apariție a parametrului x nu poate fi distinsă de simbolul mesaj_x, iar a doua nu poate fi distinsă de șirul constant 'Text x'. Aici intervine operatorul &, definiția corectă fiind:

```
irp x, < 1, 2, 3, 4 >
      mesaj_&x db 'Text &x', 0
endm
```

prin care, în primul caz, se concatenează textul fix mesaj_ cu parametrul formal x, iar în al doilea, se substituie parametrul formal x chiar dacă apare într-un șir constant de caractere.

Să considerăm un exemplu înrudit, codificat prin macroinstrucțiunea următoare:

```
genereaza macro fix, n
      X = 1
      rept n
            fix&&x db 'Text &x', 0
                  x = x + 1
            endm
endm
```

Aici e necesară concatenarea a doi parametri formali (fix și x): fiecăruia i se aplică operatorul &, la stânga sau la dreapta, după locul în care are loc concatenarea. Un apel de forma:

```
genereaza mesaj_ , 4
```

va produce același text ca macroinstrucțiunea IRP de mai sus.

5.6.2 Operatorii de literalizare șir/caracter (<>, !)

Operatorul de literalizare șir <> se utilizează atunci când se dorește ca un text în care apar eventuali separatori (spații albe, virgule etc.) să fie considerat ca un unic parametru (să fie literalizat). Operatorul se utilizează atât în definiții, cât și în invocări de macroinstrucțiuni. De exemplu, în definiția:

```
init macro x
      irp y, <x>
            db y
      endm
endm
```

dorim ca parametrul x de la nivelul exterior să fie transmis ca

atare către macroinstrucțiunea IRP. Invocarea se va face în forma:

```
init <'A', 'B', 'C', 'D'>
```

ceea ce are ca efect transmiterea listei 'A', 'B', 'C', 'D' ca un unic parametru.

Operatorul de literalizare caracter ! se aplică unui singur caracter, efectul fiind de a trata acel caracter ca un caracter obișnuit (fără a-l interpreta). Se utilizează împreună cu caractere care au semnificații speciale în macroinstrucțiuni. Să considerăm o macroinstrucțiune care definește mesaje de eroare:

```
err_gen macro N, X
err_&N db 'Eroare &N : &X', 0
endm
```

O invocare de forma:

```
err_gen 23, <Parametru ilegal>
```

se va expanda în:

```
err_23 db 'Eroare 23 : Parametru ilegal', 0
```

Dacă dorim însă să generăm un mesaj de forma 'par_1 > par_2', intrăm în conflict cu semnificația specială a caracterului '>', care intră în componența operatorului de literalizare șir. Soluția este să folosim operatorul ! și să scriem:

```
err_gen 24, <par_1 !> par_2>
```

ceea ce va genera mesajul corect:

```
err_24 24, 'Eroare 24 : par_1 > par_2', 0
```

5.6.3 Operatorul de evaluare expresie (%)

Acest operator se aplică unei expresii oarecare, efectul fiind evaluarea acelei expresii. Dintr-un anumit punct de vedere, operatorul % este inversul operatorilor de literalizare. Să considerăm macroinstrucțiunea:

```
def macro a, b
      db '&a', 0
      db '&b', 0
endm
```

care generează date. O invocare de forma:

```
alfa equ 100
beta equ 200
def <alfa + beta>, %(alfa + beta)
```

produce liniile de program:

```
db 'alfa + beta', 0
db '300', 0
```

În invocarea lui def, primul parametru este literalizat, iar al doilea evaluat, ceea ce explică textul generat.

5.7 Invocare recursivă de macroinstrucțiuni

O macroinstrucțiune se poate invoca recursiv, adică pe ea însăși. Ca și la procedurile recursive, cel mai important lucru este oprirea recursivității, care se poate face cu directivele de asamblare condiționată IFB sau IFNB. Să considerăm o macroinstrucțiune de salvare de registre în stivă. Vrem ca aceasta să permită un număr variabil de parametri. Soluția recursivă este să fixăm un număr maximal de parametri și să testăm explicit dacă primul parametru este vid:

```
push_all_1 macro r1, r2, r3, r4, r5, r6
      ifnb r1
            push r1
            push_all_1 r2, r3, r4, r5, r6
      endif
endm
```

Dacă primul parametru formal nu este vid, se generează instrucțiunea PUSH și apoi se invocă aceeași macroinstrucțiune, cu restul de parametri. Această formă poate fi folosită cu un număr oarecare de registre de la 1 la 6, de exemplu:

```
push_all_1 ax, bx, cx, dx
```

O altă variantă este cea repetitivă:

```
push_all_2 macro X
```

```
        irp y, <x>
              push y
        endm
endm
```

Invocarea necesită însă operatorul de literalizare:

```
push_all_2 <ax, bx, cx, dx>
```

5.8 Definirea macroinstrucțiunilor în macroinstrucțiuni

Se poate spune că macroinstrucțiunile automatizează oarecum procesul de definire a datelor și a instrucțiunilor. Mai mult decât atât, chiar definirea macroinstrucțiunilor se poate face prin intermediul altor macroinstrucțiuni.

Să considerăm un asemenea caz. În cazul procesorului 8086, instrucțiunile de deplasare și de rotație cu un număr de biți mai mic sau egal cu 3, se execută mai rapid ca secvențe de rotații de câte un bit, în comparație cu instrucțiunile care utilizează registrul CL. Dorim să scriem câte o macroinstrucțiune pentru cele 8 instrucțiuni de deplasare și rotație, fiecare cu doi parametri, sursa și numărul de biți, care să se expandeze în secvența optimă ca timp de execuție.

De exemplu, o invocare de forma:

```
m_shr ax, 5
```

să se expandeze în secvența:

```
mov cl, 5
shr ax, cl
```

iar o invocare de forma:

```
m_shr bx, 3
```

în secvența:

```
shr bx, 1
shr bx, 1
shr bx, 1
```

Dorim ca generarea macroinstrucțiunilor (având numele de forma m_xxx, unde xxx este numele instrucțiunii corespunzătoare) să fie făcută automat.

Începem prin a defini o macroinstrucțiune care primește (în parametrul formal operation) numele instrucțiunii respective și generează macroinstrucțiunea corespunzătoare:

```
gen macro operation
m_&operation macro operand, nr
      if nr lt 4
            rept nr
```

```
                operation operand, 1
            endm
        else
            mov cl, nr
            operation operand, cl
        endif
endm
endm
```

La o invocare de forma:

```
gen shl
```

se va genera automat macroinstrucțiunea m_shl, conform definiției echivalente:

```
m_shl macro operand, nr
      if nr lt 4
            rept nr
                  shl operand, 1
            endm
      else
            mov cl, nr
            shl operand, cl
      endif
endm
```

adică exact ce ne-am propus. Generăm acum toate cele 8 macroinstrucțiuni, observând că numele operațiilor (instrucțiunilor) respective se compun din secvențele RO, RC, SH, SA, la care se adaugă sufixele R sau L. Exploatăm acest fapt prin două macroinstrucțiuni repetitive:

```
irp X, <RO, RC, SH, SA>
      irp Y, <R, L>
            gen X&&Y
      endm
endm
```

Această secvență va defini macroinstrucțiunile m_ror, m_rol, m_rcr, m_rcl, m_shr, m_shl, m_sar și m_sal.

5.9 Tehnici avansate de utilizare a macroinstrucțiunilor

5.9.1 Macroinstrucțiuni care se autotransformă (se redefinesc) în proceduri

Un dezavantaj al utilizării intensive a macroinstrucțiunilor este consumul de memorie: dacă invocăm de 100 de ori o macroinstrucțiune, textul respectiv se va duplica de 100 de ori. Acest lucru nu deranjează în situația în care textul respectiv trebuia oricum scris (de exemplu, în secvențele de apel ale procedurilor).

Sunt însă situații în care corpul macroinstrucțiunii reprezintă o secvență oarecare de instrucțiuni. Dorim ca invocările repetate să nu conducă la repetarea corpului macroinstrucțiunii, ci la apeluri ale unei proceduri, acest lucru fiind transparent pentru utilizator. Soluția este următoarea:

```
macsub macro
       local gata
       call subr
       jmp gata
       subr proc near
              ;
              ; Corpul macroinstructiunii
              ;
              ret
       subr endp
       gata:
       macsub macro
              call subr
       endm
endm
```

Se înscrie corpul macroinstrucțiunii într-o procedură cu numele subr și se generează un apel al acestei proceduri, urmat de un salt peste procedura respectivă. Se redefinește apoi macroinstrucțiunea macsub, în așa fel încât să se genereze doar apeluri ale procedurii subr.

La prima invocare a macroinstrucțiunii macsub, se va genera textul:

```
call subr
jmp ??0000
subr proc near
       ;
       ; Corp macroinstructiune
       ;
       ret
subr endp
??0000:
```

La următoarele invocări, se va genera doar textul:

```
call subr
```

Acest exemplu arată că într-o macroinstrucțiune putem chiar redefini aceeași macroinstrucțiune.

5.9.2 Macroinstrucțiuni care generează atât date, cât și cod

În descrierea interfeței cu sistemul DOS, realizată prin fișierul header io.h (vezi 2.7 și Anexa A), s-a prezentat macroinstrucțiunea putsi, care permite afișarea (la momentul execuției) a unor șiruri constante „imediate", fără a le defini explicit într-un segment de date. Puteam deci scrie:

```
putsi <'Acesta este un mesaj la consola'>
```

fără să ne punem problema unde se memorează șirul constant respectiv. Pentru afișarea propriu-zisă, utilizăm procedura puts_proc din fișierul io.asm (vezi Anexa B), care primește în DS:SI adresa de memorie a unui șir terminat cu 0.

Problema care se pune este că definiția șirului în memorie trebuie făcută chiar în macroinstrucțiunea putsi. Vom beneficia de faptul că directivele simplificate de definire a segmentelor (în speță .code și .data) pot alterna în cuprinsul unui program. Definiția macroinstrucțiunii este următoarea:

```
putsi macro X
local string
 .data
      string db X, 0
 .code
      push   si
      lea    si, string
      call   puts_proc
      pop    si
endm
```

Se comută pe segmentul de date și se definește șirul transmis prin parametrul formal X, la care se adaugă terminatorul 0. Identificatorul string se declară local, pentru a nu fi duplicat în apeluri succesive. Se comută apoi pe segmentul de cod și se generează secvența de apel a procedurii puts_proc, cu salvarea și restaurarea registrului SI. Un apel de forma:

```
putsi <'Acesta este un mesaj la consola'>
```

se va expanda într-o secvență de forma:

```
.data
      ??0001 db 'Acesta este un mesaj la consola', 0
.code
      push si
      lea si, ??0001
      call puts_proc
      pop si
```

În mod similar, în 2.7 a fost introdusă o macroinstrucțiune geti care citește un întreg cu semn pe 16 biți de la consolă, întorcând valoarea citită în registrul AX. Problema care apare este că de la consolă se pot citi date numai la nivel de caractere.

Presupunem că dispunem de o procedură gets_proc care citește un număr limitat de caractere de la consolă. Parametrii acestei proceduri sunt adresa (near) unde se depun caracterele citite, transmisă prin SI și numărul maxim de caractere citite, transmis în CX. Citirea încetează fie la apăsarea pe Enter, fie la atingerea numărului maxim de caractere, iar după ultimul caracter se depune

terminatorul 0. Dacă CX = 0, atunci se poate introduce un număr nelimitat de caractere.

După ce s-au citit caracterele care formează numărul, acestea trebuie convertite de la șir de caractere ASCII la un întreg cu semn. Această conversie se realizează cu o procedură atoi_proc, care primește în SI adresa șirului de caractere și întoarce valoarea calculată în AX.

Macroinstrucțiunea geti se definește astfel:

```
geti macro
      local string
.data
      buffer db 8 dup (0)
.code
      push  si
      lea   si, buf
      mov   cx, 7
      call  gets_proc
      lea   si, buf
      call  atoi_proc
      pop   si
endm
```

Capitolul 6

Tehnici de programare în limbaj de asamblare

Acest capitol este dedicat tehnicilor de programare în limbaj de asamblare, adică modalităților de proiectare și implementare a modulelor de program. Deși codul mașină rezultat este mai scurt, programele sursă scrise în ASM tind să aibă dimensiuni mari. De aceea, este esențială o abordare sistematică și ordonată a dezvoltării programelor. Utilizarea disciplinată a procedurilor, a mecanismelor standard de transfer de parametri și a macroinstrucțiunilor contribuie esențial la obținerea de programe clare, eficiente și ușor de întreținut.

În esență, specificarea modulelor de program ASM nu diferă de cea specifică limbajelor de nivel înalt, în sensul că se pornește de la o descriere abstractă a algoritmului care trebuie implementat. O problemă specifică este asignarea variabilelor. Dacă într-un limbaj de nivel înalt acest lucru nu creează probleme (introducem câte variabile dorim, fără a ne pune problema spațiului alocat), în ASM trebuie să asignăm explicit variabile.

Prima modalitate este să asignăm cât mai multe variabile în registrele procesorului. Cum numărul acestora este limitat, vom fi nevoiți să asignăm variabile și în segmente de date (statice) sau în stivă. Pentru a nu crește numărul variabilelor peste o limită rezonabilă, este esențial ca modulele de program să implementeze subprobleme de dimensiuni adecvate (nu foarte mari), ceea ce implică o descompunere a problemei inițiale în subprobleme bine specificate.

6.1 Decizia simplă și decizia compusă. Evaluarea condițiilor logice

Majoritatea operațiilor de bază din programarea structurată (decizia, selecția, ciclurile cu test la partea inferioară și superioară) implică în mod inerent evaluarea unor condiții logice. În ASM, aceste condiții sunt în general de tip comparație între valori numerice.

Decizia simplă, codificată în pseudo-cod prin:

```
if (conditie)
      Ramura_if
```

se implementează în ASM prin șablonul:

```
      Evalueaza conditie
      Salt conditionat (pe conditie falsa) la eticheta_1
      ; Ramura_if
eticheta_1:
```

Decizia compusă, codificată în pseudo-cod prin;

```
      if (conditie)
            Ramura_if
      else
            Ramura_else
```

se implementează după șablonul:

```
      Evalueaza conditie
      Salt conditionat (pe conditie falsa) la et_1
      ; Ramura_if
      jmp et_2
et_1:
      ; Ramura_else
et_2:
```

Evaluarea condițiilor logice simple se realizează prin instrucțiuni de comparație, aritmetice etc., care poziționează bistabilii de condiție. De exemplu, secvența pseudo-cod:

```
      if (ax < bx)
            ; Ramura_if
      else
            ; Ramura_else
```

se implementează prin:

```
      cmp ax, bx
      jge et1:
      ; Ramura_if
      jmp et2
et1:
      ; Ramura_else
et2:
```

Evaluarea condițiilor complexe se abordează în manieră ordonată. Primul caz de bază este cel în care subcondițiile sunt conectate prin operatorul logic AND. Astfel, să considerăm o condiție logică de forma:

```
C = C1 AND C2 AND C3 AND ... AND Cn
```

și decizia compusă:

```
if (C)
      ; Ramura_if
else
```

```
            ; Ramura_else
```

Implementearea este următoarea:

```
        Evalueaza C1
        Salt conditionat (pe conditie falsa) la et_1
        Evalueaza C2
        Salt conditionat (pe conditie falsa) la et_1
        ..........................
        Evalueaza Cn
        Salt conditionat (pe conditie falsa) la et_1
        ;
        ; Ramura_if
        ;
        jmp et_2
et_1:
        ;
        ; Ramura_else
        ;
et_2:
```

Să considerăm, de exemplu, secvența pseudo-cod:

```
        if (car >= '0' AND car <= '9') {
              sir [i] = car - '0';
              i = i + 1;
        }
```

în care presupunem că variabila car se află în AL, iar indicele i în registrul BX. Implementarea este următoarea:

```
        cmp   al, '0'           ; Evaluare al >= '0'
        jb    et_1              ; Salt pe cond. falsa (al < '0')
        cmp   al, '9'           ; Evaluare al <= '9'
        ja    et_1              ; Salt pe cond. falsa (al > '9')
        sub   al, '0'           ; Calcul car - '0'
        mov   sir [bx], al      ; Depunere in sir [i]
        inc   bx                ; Incrementare i
et_1:
```

Al doilea caz de bază este cel în care subcondițiile sunt conectate prin operatorul OR. Să considerăm condiția compusă:

```
        C = C1 OR C2 OR C3 OR ... OR Cn
```

și aceeași formă de decizie compusă:

```
        if (C)
              ; Ramura_if
        else
              ; Ramura_else
```

Implementarea este următoarea:

```
        Evalueaza C1
        Salt conditionat (pe conditie adevarata) la et_1
        Evalueaza C2
        Salt conditionat (pe conditie adevarata) la et_1
        ..........................
        Evalueaza Cn
        Salt conditionat (pe conditie adevarata) la et_1
        ;
        ; Ramura_else
        ;
        jmp et_2
et_1:
        ;
        ; Ramura_if
        ;
et_2:
```

Cele două șabloane de implementare pentru condiții compuse de tip AND și OR se bazează pe proprietățile elementare ale operațiilor logice respective. Astfel, la operația AND, e suficient ca un singur operand (o subcondiție) să fie fals, pentru ca întreaga condiție să fie falsă. Similar, la operația OR, e suficient ca un singur operand să fie adevărat, pentru ca întreaga condiție să fie adevărată.

A treia schemă de dezvoltare se referă la implementarea condițiilor negate. O secvență pseudo-cod de forma:

```
        if (NOT conditie)
                ; Ramura_if
        else
                ; Ramura_else
```

se implementează la fel cu schema if-else obișnuită, dar cu inversarea saltului condiționat:

```
        Evalueaza conditie
        Salt conditionat (pe conditie adevarata) la et_1
        ;
        ; Ramura_if
        ;
        jmp et_2
        et_1:
        ;
        ; Ramura_else
        ;
et_2:
```

Cu aceste trei operații de bază (AND, OR, NOT), putem acum evalua orice tip de condiție logică. Să considerăm secvența pseudo-cod:

```
        if ((C1 AND C2) OR C3)
                ; Ramura_if
        else
                ; Ramura_else
```

Considerăm subcondițiile C1 AND C2 și C3 și aplicăm pentru început șablonul de la operația OR:

```
      Evalueaza (C1 AND C2)
      Salt conditionat (pe conditie adevarata) la et_1
et_3:
      Evalueaza (C3)
      Salt conditionat (pe conditie adevarata) la et_1
      ;
      ; Ramura_else
      ;
      jmp et_2
et_1:
      ;
      ; Ramura_if
      ;
et_2:
```

Detaliem acum evaluarea subcondiției (C1 AND C2), observând că se sare la eticheta et_1 dacă ambele subcondiții C1 și C2 sunt adevărate; altfel se sare la eticheta et_3:

```
      Evalueaza C1
      Salt conditionat (pe conditie falsa) la et_3
      Evalueaza C2
      Salt conditionat (pe conditie adevarata) la et_1
et_3:
      Evalueaza (C3)
      Salt conditionat (pe conditie adevarata) la et_1
      ;
      ; Ramura_else
      ;
      jmp et_2
et_1:
      ;
      ; Ramura_if
      ;
et_2:
```

După aceste modele, se pot evalua pas cu pas condiții oricât de complicate. Șabloanele de evaluare se utilizează și la celelalte operații din programarea structurată.

6.2 Cicluri cu test la partea superioară și inferioară

Ciclul cu test la partea superioară, descris în pseudo-cod prin:

```
while (conditie)
      ; Bloc
```

se implementează după șablonul:

```
et_1:
      Evalueaza conditie
      Salt conditionat (pe conditie falsa) la et_2
      ;
      ;
      ; Bloc
      jmp et_1
```

Să considerăm secvența următoare în limbajul C:

```
      while (*s >= '0' && *s <= '9') {
            n = 10 * n + *s - '0';
            s++;
      }
```

unde s este un pointer la char, iar n un întreg. Această secvență este tipică pentru conversia ASCII-întreg. Presupunem variabila n asignată în AX, iar pointerul s asignat (ca adresă near) în registrul SI și aplicăm șablonul de implementare:

```
et_1:
      Evalueaza ( [SI] >= '0')
      Salt pe conditie falsa la et_2
      Evalueaza ( [SI] <= '9')
      Salt pe conditie falsa la et_2
      AX = AX * 10 + [SI] - '0'
      SI = SI + 1
      jmp et_1
et_2:
```

În codificarea propriu-zisă, se va încărca [si] în registrul al pentru o operare mai eficientă:

```
      mov bx, 10
et_1:
      mov   cl, [si]
      xor   ch, ch              ; CX <-- caracterul de la adresa SI
      cmp   dl, '0'             ; Prima subconditie
      jb    et_2                ; Salt pe conditie falsa
      cmp   dl, '0'             ; A doua subconditie
      ja    et_2                ; Salt pe conditie falsa
      mul   bx                  ; n = n * 10
      sub   cl, '0'             ; *s - '0'
      add   ax, cx              ; Valoare finala n
      inc   si                  ; s++
      jmp   et_1
```

Ciclurile cu test la partea inferioară, descris în pseudo-cod prin una din formele:

```
do                                repeat
      ; Bloc                            ; Bloc
while (conditie)                  until (conditie)
```

se implementează după șabloanele:

```
et:                                          et:
; Bloc                                       ; Bloc
Evalueaza conditie                           Evalueaza conditie
Salt pe conditie adevarata la et             Salt pe conditie falsa la et
```

Să considerăm un algoritm tipic pentru conversia întreg-ASCII, descris în limbajul C:

```
do {
      *s++ = n % 10 + '0';
      n = n/10;
} while (n != 0);
*s = 0;
```

Prin împărțiri succesive la 10 se generează cifrele corespunzătoare întregului n și se depun în șirul de caractere s (cifrele rezultă în ordine inversă). Implementarea este următoarea (considerăm n memorat în registrul AX, iar pointerul n în registrul index DI):

```
      mov   bx, 10
et:
      xor   dx, dx              ; Deimpartit = DX:AX
      div   bx                  ; AX = n / 10, DX (DL) = n % 10
      add   dl, '0'             ; n % 10 + '0'
      mov   [di], dl            ; Depunere in *s
      inc   di                  ; si apoi incrementare s
      test  ax, ax              ; Compara n cu 0
      jnz   et                  ; Reluare ciclu
      mov   byte ptr [di], 0    ; *s = 0
```

Un caz particular de ciclu cu test la partea superioară este ciclul cu contor ascendent, descris în pseudo-cod prin:

```
for (contor = vi to vf step pas)
      ; Bloc
```

în care se consideră că pas este o valoare strict pozitivă. Dacă specificația pasului lipsește, se consideră implicit pasul 1. Această descriere se poate detalia într-un ciclu de tip while și o inițializare:

```
contor = vi;
 while (contor <= vf) {
 ; Bloc
 contor = contor + pas;
}
```

ceea ce arată că se poate aplica șablonul de implementare de la ciclul while.

Ciclul cu contor descendent, descris în pseudo-cod prin:

```
for (contor = vi downto vf step pas)
 ; Bloc
```

în care, de asemenea, se consideră că pas este pozitiv și implicit 1. Și această formă se poate detalia în:

```
 contor = vi;
        while (contor >= vf) {
; Bloc
       contor = contor - pas;
}
```

Să considerăm, de exemplu, o secvență tipică de translatare la dreapta cu o poziție a elementelor unui tablou de întregi:

```
for (i = 99 downto 1)
      TAB [i] = TAB [i-1];
```

Asignăm variabila i la registrul SI. Trebuie observat că indicii variază după elementele tabloului, dar adresele variază cu câte 2 octeți la fiecare element. Pentru o implementare ordonată, vom varia variabila indice SI exact ca în specificarea algoritmului și o vom ajusta temporar prin înmulțire cu 2 înaintea accesării elementelor tabloului. Detalierea ciclului este:

```
i = 99;
et_1:
 Evalueaza ( i >= 1)
 Salt pe conditie falsa la et_2
 TAB [i] = TAB [i-1];
 i = i - 1;
et_2:
```

După această detaliere, implementarea devine de rutină:

```
 mov di, 99              ; i = 99;
et_1:
 cmp di, 1               ; Evalueaza (i >= 1)
 jl et_2                 ; Salt pe conditie falsa (i < 1)
 shl di, 1               ; Temporar, DI <-- 2*DI
 mov ax, TAB[si]         ; TAB [i]
 mov TAB[si-2], ax       ; TAB [i-1]
 shr di, 1               ; Refacere DI
 dec di                  ; i = i - 1
et_2:
```

Secvența de mai sus ar putea fi implementată și prin instrucțiuni specifice șirurilor de cuvinte (se presupune că registrele DS și ES indică segmentul în care este memorat tabloul TAB):

```
lea si, TAB [98*2]      ; Sursa initiala
lea di, TAB [99*2]      ; Destinatie initiala
std                     ; Sens descendent
mov cx, 99              ; Numar iteratii
rep movsw               ; Transfer
```

O altă formă posibilă, care permite generalizări interesante este copierea la nivel de octet:

```
lea si, TAB [98*2 + 1] ; incepem de la
lea di, TAB [99*2 + 1] ; ultimul octet
std
mov cx, 99*2           ; Numar de octeti
rep movsb              ; Copiere
```

În general, prelucrarea tablourilor de tip oarecare presupune înmulțirea indicilor de acces cu numărul de octeți ai tipului de bază al tabloului. În acest sens, inițializările registrelor SI și DI din ultima secvență se pot generaliza astfel:

```
lea si, TAB [98 * (TYPE TAB) + (TYPE TAB - 1)]
lea di, TAB [99 * (TYPE TAB) + (TYPE TAB - 1)]
```

Dacă secvența de copiere propriu-zisă se înlocuiește cu:

```
mov cx, 99 * (TYPE TAB)
rep movsb
```

atunci se obține o secvență care implementează algoritmul dat, indiferent de tipul tabloului. Să considerăm un tablou de structuri de tipul:

```
MY_STRUC struc
 n dw ?
 text db    10 dup (0)
ENDS
```

și definim un tablou de 20 de structuri:

```
.data
 tab MY_REC 20 dup (< , >)
```

Secvența de translatare se poate scrie:

```
lea si, tab [(LENGTH tab - 1) * (TYPE MY_REC) - 1]
lea di, tab [(LENGTH tab ) * (TYPE MY_REC) - 1]
std
mov cx, (LENGTH tab - 1) * (TYPE MY_REC)
rep movsb
```

prin care se poziționează DI (destinația) pe ultimul octet al ultimei înregistrări din tablou, iar SI (sursa) pe ultimul octet al penultimului element din tablou. Numărul de iterații la nivel de elemente este numărul de elemente al tabloului, micșorat cu o unitate; la nivel de octeți, se înmulțește această valoare cu dimensiunea unui element.

O altă formă posibilă de scriere exploatează legătura dintre operatorii LENGTH, TYPE și SIZE. De exemplu, inițializarea contorului CX s-ar mai putea scrie:

```
mov cx, SIZE tab - TYPE tab
```

În cazurile în care ciclul cu contor se poate descrie prin:

```
for (contor = n downto 1)
        ; Bloc
```

sau atunci când se repetă de n ori o anumită operație, iar valoarea curentă a contorului nu contează, ciclul se poate implementa prin instrucțiunea LOOP:

```
mov cx, n
et:
        ; Bloc
loop et
```

Să considerăm o secvență de determinare a maximului și a minimului unui tablou de întregi cu semn, definit prin:

```
.data
 TABLOU      dw 100 dup(?)
 n           dw ($-TABLOU)/2
 val_max     dw ?
 val_min     dw ?
 i_max       dw ?
 i_min       dw ?
```

Se dorește determinarea atât a valorilor maxime și minime, cât și a indicilor pe care apar aceste elemente.

Secvența se poate descrie în pseudo-cod prin:

```
val_max = TABLOU [0];
val_min = TABLOU [0];
i_max = 0;
i_min = 0;
for (i = 1 to n-1) {
      if (TABLOU [i] > val_max) {
            val_max = TABLOU [i];
            i_max = i;
      }
      else if (TABLOU [i] < val_min) {
            val_min = TABLOU [i];
            i_min = i;
      }
}
```

Pentru implementare, vom presupune că adresa elementului TABLOU[i] este alocată în registrul BX. Se va incrementa direct această adresă, iar ciclul va fi implementat printr-o instrucțiune loop. Indicele elementului curent se obține printr-o diferență între adresa curentă (BX) și adresa de început a tabloului (SI) și o împărțire la 2.

```
.code
       lea bx, TABLOU    ; Adresa elementului TABLOU [0]
       mov si, bx        ; Copiata si în SI
aici_1:
```

```
        mov cx, n
        dec cx              ; Sunt n-1 iteratii
aici_2:
        mov ax, [bx]
        mov val_max, ax     ; val_max = TABLOU [0]
        mov val_min, ax     ; val_min = TABLOU [0]
        mov i_max, 0        ; i_max = 0
        mov i_min, 0        ; i_min = 0
        add bx, 2           ; Adresa elementului TABLOU [1]
et_1:
        mov ax, [bx]        ; AX <-- TABLOU [i]
        cmp ax, val_max     ; Evalueaza (TABLOU [i] > val_max)
        jle et_2            ; Salt pe conditie negata
        mov val_max, ax     ; val_max = TABLOU [i]
        push bx             ; Salvare adresa curenta
        sub bx, si          ; Adr. curenta - adr. de inceput
        shr bx, 1 ; / 2
        mov i_max, bx       ; = indicele i_max
        pop bx              ; Refacere adresa curenta
et_2:
        cmp ax, val_min     ; Evalueaza (TABLOU [i] < val_min)
        jge et_3            ; Salt pe conditie negata
        mov val_min, ax     ; val_min = TABLOU [i]
        push bx             ; Similar
        sub bx, si
        shr bx, 1
        mov i_min, bx
        pop bx
et_3:
        add bx, 2           ; Adresa elementului urmator
        loop et_1           ; Ciclu dupa CX
```

Se observă că implementarea ciclului prin instrucțiunea LOOP complică determinarea indicelui elementului curent. Dacă s-ar fi implementat un ciclu ascendent obișnuit, înlocuind secvența dintre etichetele aici_1 și aici_2 prin:

```
aici_1:
        mov cx, 1
        cmp cx, n
        jl et_4
aici_2:
```

iar secvența de după eticheta et_3 prin:

```
et_3:
        add bx, 2
        inc cx
        jmp et_1
et_4:
```

atunci, la fiecare iterație, indicele elementului curent ar fi fost disponibil în registrul CX, iar secvențele de determinare a indicilor i_max și i_min s-ar fi redus la simple transferuri de forma:

```
mov i_min, cx
```

Exemplul de mai sus arată că implementarea ciclurilor cu contor prin instrucțiunea LOOP, aparent mai simplă, poate conduce la complicații în interiorul ciclului.

Există și cicluri cu mai multe puncte de ieșire (o ieșire normală și una sau mai multe ieșiri forțate), ca în secvența pseudo-cod următoare, în care prin break s-a marcat ieșirea forțată din ciclu:

```
while (conditie_1) {
; Bloc_1
if (conditie_2)
break;
; Bloc_2
}
```

O asemenea situație se implementează combinând șabloanele de la while și if:

et_1:

```
Evalueaza (conditie_1)
Salt pe conditie falsa la et_2
; Bloc_1
Evalueaza (conditie_2)
Salt pe conditie adevarata la et_2
; Bloc_2
jmp et_1
et_2:
```

6.3 Selecția. Tabele de salt sau de apel de proceduri

Operația de selecție se descrie în pseudo-cod prin:

```
selecteaza (c) dintre {
c1: Bloc_1;
c2: Bloc_2;
.....
cn: Bloc_n;
[default: Bloc_d;]
}
```

în care c1, c2, ..., cn sunt așa-numitele cazuri (case). Acestea sunt, de fapt, valori de același tip cu variabila c. Cazul default corespunde situației în care variabila c nu are nici una din valorile c1, c2, ..., cn și este opțional. Blocurile de instrucțiuni Bloc_1, Bloc_2, ..., Bloc_n pot fi și vide.

Implementarea naturală a selecției pornește de la observația că o asemenea operație este echivalentă cu o succesiune de decizii, după cum urmează:

```
if (c = c1)
 ; Bloc_1
 else if (c = c2)
```

```
; Bloc_2
.................
else if (c = cn)
; Bloc_n
else
; Bloc_d
```

Se pot aplica acum șabloanele de implementare de la operația de decizie, ceea ce înseamnă comparații succesive și salturi condiționate. Această soluție de implementare devine incomodă atunci când numărul cazurilor este mare.

O altă soluție de implementare, mult mai eficientă, utilizează tabele de salt sau de apel de proceduri. Să presupunem că blocurile de instrucțiuni sunt organizate în felul următor:

```
et_1:
; Bloc_1
jmp et
et_2:
; Bloc_2
jmp et
.................
et_n:
; Bloc_n
jmp et
et_d:
  ; Bloc_d
et:
```

deci ieșirea din operația de selecție se face pe la eticheta et.

Ideea soluției de implementare este definirea unei tabele de salt, inițializată cu punctele de intrare în blocurile de instrucțiuni (deci cu adresele etichetelor et_i) și calculul automat al adresei corespunzătoare de salt, pe baza căutării valorii curente c într-un tabel de cazuri posibile.

Pentru a fixa ideile, presupunem variabila c și cazurile c1,c2, ..., cn ca fiind reprezentabile pe câte un octet; de asemenea, presupunem toate blocurile de instrucțiuni definite în același segment de cod; registrele DS și ES indică segmentul curent de date. Variabilele n și c au semnificațiile din descrierea operației (variabila de selecție, respectiv numărul de cazuri posibile). Definiția celor două tabele este:

```
.data
 case db c1, c2, c3, ..., cn
 tabjmp dw et_1, et_2, ..., et_n
 c db ?
 n dw ?
```

Să considerăm un exemplu concret. Se citește un caracter de la tastatură și, funcție de valoarea sa, se afișează un mesaj la consolă. Considerăm numărul cazurilor posibile ca fiind 4, iar constantele de selecție ca fiind caracterele 'a', 'b',

'c' și 'd'. Citirea caracterelor se execută într-o buclă din care se iese la apăsarea tastei Enter. Implementarea este următoarea:

```
.model large
      include io.h
.stack 1024
.data
      case   db 'a', 'b', 'c', 'd'
      tabjmp dw et_1, et_2, et_3, et_4
      n      dw 4
.code
start:
      init_ds_es          ; Initializare DS si ES
iar:
      getc                ; Citire caracter in AL
      cmp    al, cr       ; Este Enter ?
      je     gata         ; Daca da, oprire
      lea    di, case     ; Adresa tabela de cazuri
      mov    cx, n        ; Numar de cazuri explicite
      cld                 ; Directie ascendenta
      repne  scasb        ; Cautare caz (AL) in tabela
      jne    et_d         ; Daca ZF = 0, inseamna ca nu s-a
                          ; identificat nici un caz explicit
                          ; deci este cazul default
      dec    di           ; S-a gasit un caz explicit
                          ; DI este pozitionat pe octetul
                          ; urmator, asa ca il decrementam
      lea    bx, case     ; Adresa tabela de cazuri
      sub    di, bx       ; Diferenta = deplasament, in
                          ; gama: 0 ... n-1
      shl    di, 1        ; inmultire cu 2 (adrese pe word)
      jmp tabjmp [di]     ; Salt indirect la cazul respectiv
gata:
      exit_dos
      ;
      ; Blocurile de instructiuni
      ; care trateaza
      ; cazurile explicite si implicite
                          ;
et_1:
      putsi <cr, lf, 'Cazul 1 (a)', cr, lf>
      jmp et
et_2:
      putsi <cr, lf, 'Cazul 2 (b)', cr, lf>
      jmp et
et_3:
      putsi <cr, lf, 'Cazul 3 (c)', cr, lf>
      jmp et
et_4:
      putsi <cr, lf, 'Cazul 4 (d)', cr, lf>
      jmp et
et_d:
      putsi <cr, lf, 'Cazul default', cr, lf>
et:
      ;
```

```
; Punct de iesire din
; operatia de selectie
;
jmp iar
end start
```

Căutarea în tabela case se face prin instrucțiuni cu șiruri de caractere, după modelul standard descris la aceste instrucțiuni. Din adresa elementului identificat în tabelă se calculează indicele acestuia (de la 0 la n-1) și, prin înmulțire cu 2, poziția corespunzătoare din tabela de adrese de salt. Se execută apoi un salt indirect intrasegment.

Alte variante posibile de implementare sunt:

- cazurile memorate pe mai mult de un octet - se definește tabela case în mod corespunzător, și se execută o secvență de căutare explicită, după modelul:

```
for (i = 0 to n-1)
     if (case [i] = c)
          break;
     if (i < n)
          jmp tabjmp [i]
     else
          jmp et_d
```

- blocurile asociate cazurilor sunt în segmente de cod diferite - se definește tabela de salt cu adrese de tip far, iar etichetele et_i și et_d se definesc cu directiva LABEL și atributul FAR;
- blocurile asociate cazurilor sunt implementate ca proceduri - se definește tabela de adrese inițializată cu numele procedurilor respective și se înlocuiește instrucțiunea de salt indirect cu una de apel indirect de procedură; punctul de ieșire din operația de selecție va fi cel imediat următor apelului indirect de procedură.

6.4 Transferul parametrilor către proceduri

Proiectarea ordonată și sistematică a procedurilor este un punct cheie în dezvoltarea unui sistem de programe în limbaj de asamblare. Problemele de bază care trebuie urmărite sunt:

- transferul parametrilor;
- întoarcerea rezultatelor;
- zone de date proprii procedurilor;
- controlul stivei;
- recursivitatea;
- proceduri cu număr variabil de parametri.

Pentru fiecare din aceste probleme, există tehnici sistematice de abordare, care vor fi prezentate în continuare.

Prima problemă se referă la transferul parametrilor către proceduri. Dacă în limbajele de nivel înalt acest lucru este impus de sintaxa limbajului (se definește o listă de parametri), în limbaj de asamblare există multiple posibilități de transfer. Aceasta este, de altfel, și cauza pentru care o proiectare nesistematică a procedurilor, poate conduce la programe greu de citit și înțeles, predispuse la erori și foarte greu de întreținut.

Pe de altă parte, o proiectare îngrijită a procedurilor, combinată eventual cu macroinstrucțiuni adecvate, care respectă tehnicile standard de transmitere a parametrilor, contribuie esențial la dezvoltarea unor programe ușor de înțeles și de întreținut, asemănătoare - din acest punct de vedere - programelor în limbaje de nivel înalt.

6.4.1 Tipuri de transfer (prin valoare sau prin referință)

O primă chestiune care trebuie decisă în proiectarea unei proceduri este tipul de transfer al parametrilor. Se pot utiliza două asemenea tipuri:

- transfer prin valoare, care implică transmiterea conținutului unei variabile;
- transfer prin referință, care implică transmiterea adresei de memorie a unei variabile.

Alegerea între aceste două tipuri de transfer se poate face după următoarele criterii:

- dacă variabila care trebuie transmisă nu este alocată în memorie, ci într-un registru, se va alege transmiterea prin valoare;
- structurile de date de volum mare (tablouri, structuri, tablouri de structuri etc.) vor fi transmise totdeauna prin referință;
- dacă procedura trebuie să modifice o variabilă parametru formal, care este alocată în memorie, se va alege transferul prin referință; (modificarea unui parametru formal din interiorul procedurii trebuie totuși utilizată cât mai puțin, deoarece este o cauză majoră de erori; este preferabil un transfer prin valoare și întoarcerea valorii modificate).

Pentru a fixa ideile, să considerăm o procedură pro_add, de tip near, care adună două numere pe 32 de biți, întorcând rezultatul în perechea de registre DX:AX. Datele sunt memorate în variabilele n1 și n2, iar rezultatul în variabila rez:

```
.data
 n1   dd 10000H
 n2   dd 20000H
 rez  dd ?
```

Să presupunem că transmitem parametrii prin registre, ceea ce înseamnă că avem nevoie de 4 registre generale, de exemplu DX:AX pentru primul parametru și CX:BX pentru al doilea. Secvența de apel a procedurii este următoarea:

```
.code
      mov ax, word ptr n1
```

```
mov dx, word ptr n1 + 2 ; DX:AX = primul parametru
mov bx, word ptr n2
mov cx, word ptr n1 + 2 ; CX:BX = al doilea parametru
call near ptr pro_add
mov word ptr rez, ax    ; Rezultat în DX:AX
mov word ptr rez + 2, dx
```

Procedura pro_add se detaliază astfel:

```
pro_add proc near
      add ax, bx
      adc dx, cx
      ret
pro_add endp
```

Să considerăm acum varianta transmiterii prin referință a parametrilor, în care se transmit către procedură adresele de tip near ale variabilelor n1 și n2, prin registrele SI și DI. Secvența de apel este:

```
lea si, n1
lea di, n2
call near ptr pro_add
mov
mov word ptr rez, ax    ; Rezultat in DX:AX
mov word ptr rez + 2, dx
```

Procedura se dezvoltă în felul următor:

```
pro_add proc near
      mov ax, [si]      ; Partea low din primul numar
      add ax, [di]      ; + partea low din al doilea
      mov dx, [si+2]    ; Partea high din primul numar
      adc dx, [di+2]    ; + partea high din al doilea
      ret
pro_add endp
```

În esență, se vede că transmiterea prin referință implică operații de adresare indirectă în interiorul procedurii.

Un aspect important este salvarea și restaurarea registrelor implicate în transferul parametrilor, ca și a celor utilizate în interiorul procedurii.

6.4.2 Transfer prin registre

Vom trece acum la analiza modalităților efective de transfer a parametrilor, fie ei valori sau adrese. O primă modalitate este transferul prin registrele mașinii. Avantajul acestei soluții este faptul că, în procedură, parametrii actuali sunt disponibili imediat.

Pentru conservarea registrelor, acestea se salvează în stivă înainte de apel și se refac după revenirea din procedură. Secvența de apel este, deci, organizată după șablonul:

```
; Salvare in stiva a registrelor implicate in transfer
; incarcare registre cu parametrii actuali
; Apel de procedura
; Refacere registre din stiva
```

Dezavantajele acestei modalități sunt:

- numărul limitat de registre ale mașinii - e posibil să existe registre ocupate sau pur și simplu să fie mai mulți parametri decât registre disponibile;
- neuniformitatea metodei - nu există o modalitate ordonată de transfer, fiecare procedură având propriile reguli de transfer.

6.4.3 Transfer prin zonă de date

În această variantă, se pregătește anterior o zonă de date și se transmite către procedură adresa acestei zone de date. În această formă, organizarea zonei de date și secvența de apel a procedurii pro_add din paragraful anterior este:

```
.data
      zona  label dword
      n1    dd 10000H
      n2    dd 20000H
      rez   dd ?
.code
      lea   bx, zona
      call  pro_add
```

Procedura pro_add se scrie acum în forma:

```
pro_add proc near
      mov ax, [bx]        ; Partea low din primul numar
      add ax, [bx+4]      ; + partea low din al doilea
      mov dx, [bx+2]      ; Partea high din primul numar
      adc dx, [bx+6]      ; + partea high din al doilea
      ret
pro_add endp
```

Pentru un acces comod la zona de parametri, se recomandă definirea unei structuri care să descrie organizarea zonei de date:

```
TIP_ZONA struc
       nr1 dd ?
       nr2 dd ?
       rez dd ?
TIP_ZONA ends

.data
      zona TIP_ZONA <10000, 20000, ?>
.code
      lea bx, zona
      call pro_add
```

Procedura pro_add se scrie astfel:

```
pro_add proc near
     mov ax, [bx].nr1          ; Partea low din primul numar
     add ax, [bx].nr2          ; + partea low din al doilea
     mov dx, [bx].nr1+2 ;      ; Partea high din
                               ; primul numar
     adc dx, [bx+6].nr2+2      ; + partea high din al doilea
     ret
pro_add endp
```

6.4.4 Transfer prin stivă. Descărcarea stivei

Transferul parametrilor prin stivă este cea mai importantă modalitate de transfer. Avantajele acestei metode sunt uniformitatea (se asigură un mecanism unic de transfer pentru toate procedurile) și compatibilitatea cu limbajele de nivel înalt (majoritatea compilatoarelor utilizează această metodă). Transferul prin stivă este obligatoriu în situația în care aplicația conține atât module în ASM, cât și module în limbaj de nivel înalt.

În principiu, transferul prin stivă constă în plasarea în stivă (prin instrucțiuni de tip PUSH) a parametrilor, înainte de apelul procedurii. Astfel, procedura va găsi parametrii în stivă, imediat după adresa de revenire.

Problemele de bază care trebuie avute în vedere la implementarea acestui tip de transfer sunt:

- tipul procedurii (FAR sau NEAR);
- tipul parametrilor, în special al celor de tip adresă (FAR sau NEAR);
- ordinea de plasare a parametrilor în stivă;
- accesul la parametri din interiorul procedurii;
- descărcarea stivei (de către programul apelant sau de către procedură).

Tipul procedurii, tipul parametrilor și ordinea lor sunt importante pentru calculul deplasamentelor în stivă și pentru accesul corect la parametri.

Tehnica de acces standard la parametrii procedurii se bazează pe adresarea bazată (eventual și indexată) prin registrul BP, care presupune registrul SS ca registru implicit de segment. Accesul se realizează prin operațiile următoare, efectuate chiar la intrarea în procedură:

- se salvează BP în stivă;
- se copiază SP în BP;
- se salvează în stivă (eventual) registrele utilizate de procedură;
- se accesează parametrii prin adresare indirectă cu BP.

La încheierea procedurii, se execută operațiile următoare:

- se refac registrele salvate;
- se reface BP;
- se revine în programul apelant prin RET.

Să considerăm aceeași procedură pro_add (de data aceasta de tip FAR), implementată prin această tehnică. Secvența de apel va fi:

```
.data
      n1    dd 10000H
      n2    dd 20000H
      rez   dd ?
.code
      push word ptr n1+2        ; Partea high la adrese mari
      push word ptr n1          ; Partea low la adrese mici
      push word ptr n2+2        ; Similar pentru n2
      push word ptr n2
      call far ptr pro_add      ; Apel
      add  sp, 8                ; Descarcare stiva
      mov  word ptr rez, ax     ; Depunere
      mov  word ptr rez+2, dx ; rezultat
```

Plasarea datelor în stivă trebuie să țină seama de modul de reprezentare în memorie. Astfel, numerele pe 32 de biți se plasează în stivă în așa fel încât la adrese mici să se găsească partea mai puțin semnificativă. De asemenea, se observă descărcarea stivei (refacerea registrului SP la valoarea dinaintea secvenței de apel), prin adunarea explicită la SP a numărului de octeți care a fost plasat în stivă.

Pentru a accesa corect parametrii în stivă, este bine să figurăm imaginea stivei după intrarea în procedură și salvarea registrului BP, ținând cont de tipul procedurii, de numărul, tipul și ordinea parametrilor în stivă. Această imagine este ilustrată în Figura 6.1 (reamintim că, în reprezentarea grafică a spațiului de memorie, adresele cresc de sus în jos).

Procedura pro_add se scrie în felul următor:

```
pro_add proc far
        push bp             ; Secventa tipica
        mov bp, sp          ; de acces
        mov ax, [bp+10]     ; n1 low
        add ax, [bp+6]      ; n2 low
        mov dx, [bp+12]     ; n1 high
        adc dx, [bp+8]      ; n2 high
        pop bp              ; Refacere bp
        ret                 ; Revenire
pro_add endp
```

Calculul explicit al deplasamentelor parametrilor (de tipul [bp+8], [bp+10] etc.) reprezintă o sursă potențială de greșeli. În plus, întreținerea procedurii este foarte greoaie. Dacă se schimbă tipul procedurii din FAR în NEAR sau dacă se schimbă ordinea celor doi parametri, toate liniile de program care conțin deplasamente de acest gen trebuie rescrise.

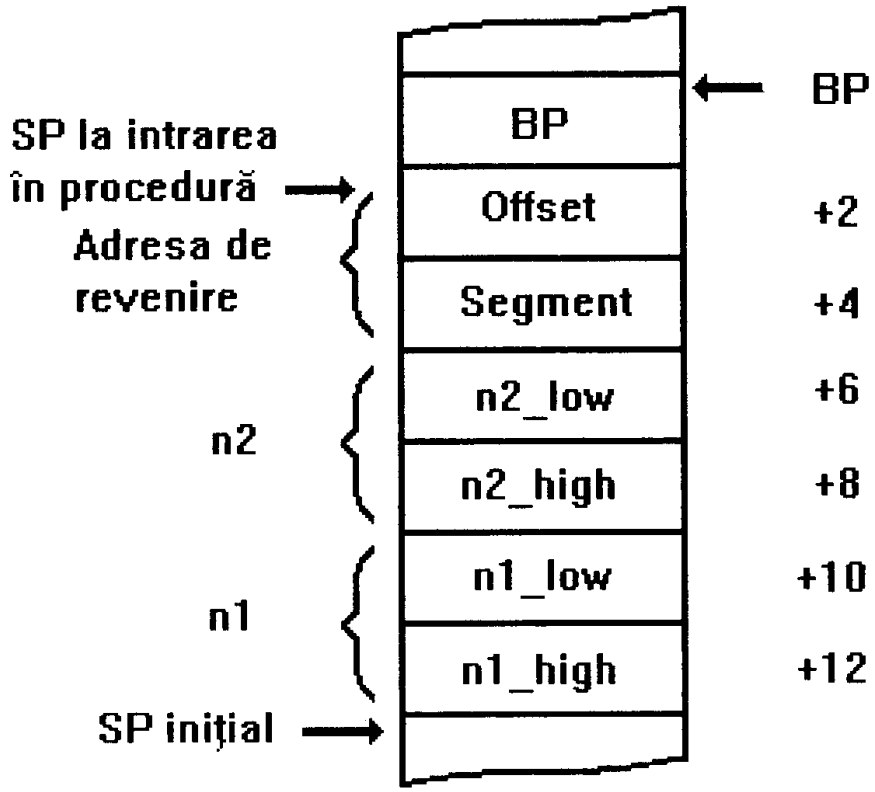


Figura 6.1 Imaginea stivei la intrarea în procedura pro_add

Aceste probleme se rezolvă elegant prin definirea unei structuri șablon care să conțină imaginea stivei, de la registrul BP în jos. Dacă am figurat grafic imaginea stivei, definirea structurii șablon este imediată (vezi Figura 6.1):

```
sablon_1 struc
      _bp          dw ?                ; BP
      _cs_ip       dw 2 dup (?)        ; Adresa de revenire
      n2_low       dw ?                ;
      n2_high      dw ?                ; Parametri
      n1_low       dw ?                ;
      n1_high      dw ?                ;
sablon_1 ends
```

Procedura se rescrie acum în forma:

```
pro_add proc far
       push bp
       mov  bp, sp
       mov  ax, [bp].n1_low
       add  ax, [bp].n2_low
       mov  dx, [bp].n1_high
       adc  dx, [bp].n2_high
```

```
        pop bp
        ret
pro_add endp
```

ceea ce este mult mai clar decât versiunea anterioară. În plus, dacă se modifică tipul procedurii sau ordinea parametrilor, nu trebuie modificată decât definiția structurii șablon; asamblorul va calcula corect noile deplasamente.

În forma de mai sus, procedura pro_add corespunde unei funcții C cu prototipul:

```
long _pro_add (long x1, long x2);
```

Să presupunem acum că procedura este fără tip (nu întoarce nimic), dar în lista de parametri se transmite adresa rezultatului, ceea ce ar corespunde unui prototip C de forma:

```
void _pro_add(long x1, long x2, long *adr_rez);
```

Dacă presupunem că adresa rezultatului este de tip FAR, secvența de apel a procedurii va fi:

```
push ax                     ; Salvare temporara AX
                            ;
                            ; Aici incepe secventa de apel
                            ;
push word ptr n1+2          ; Partea high la adrese mari
push word ptr n1            ; Partea low la adrese mici
push word ptr n2+2          ; Similar pentru n2
push word ptr n2
mov  ax, SEG rez            ; Adresa de segment
push ax                     ; La 286 se poate direct:
                            ; push SEG rez
mov  ax, OFFSET rez         ; Offset
push ax
call far ptr pro_add        ; Apel
add  sp, 12                 ; Descarcare stiva
                            ;
                            ; Aici se termina secventa de apel
                            ;
pop ax                      ; Refacere AX
```

Plasarea unor adrese FAR în stivă trebuie să țină seama de modul de reprezentare al pointerilor de tip FAR (definiți de exemplu prin directiva Define DoubleWord): la adrese mici se memorează offset-ul, iar la adrese mari, adresa de segment. Dacă este cazul, se salvează în stivă registrele utilizate în secvența de apel (în cazul de față, AX).

Structura de tip șablon de acces se rescrie, adăugând noul parametru de tip adresă:

```
sablon_2 struc
        _bp         dw ?          ; BP
        _cs_ip      dw 2 dup (?); Adresa de revenire
```

```
        adr_rez         dd ?   ; Adresa rezultat (FAR)
        n2_low          dw ?   ;
        n2_high         dw ?   ; Parametri
        n1_low          dw ?   ;
        n1_high         dw ?   ;
sablon_2 ends
```

În această variantă, procedura pro_add este:

```
pro_add proc far
      push bp                  ; Secventa tipica de acces
      mov bp, sp
                               ;
      push ax                  ;
      push bx                  ; Salvari registre utilizate
      push es                  ;
                               ;
      les bx, [bp].adr_rez     ; Pointer la rezultat
      mov ax, [bp].n1_low      ;
      add ax, [bp].n2_low      ; Calcul parte low
      mov es:[bx], ax          ; Depunere rezultat low
      mov ax, [bp].n1_high     ;
      adc ax, [bp].n2_high     ; Calcul parte high
      mov es:[bx+2], ax        ; Depunere rezultat high
                               ;
      pop es                   ;
      pop bx                   ; Refacere registre
      pop ax                   ;
                               ;
      pop bp                   ; Refacere BP
      ret                      ; Revenire
pro_add endp
```

În procedură nu putem face presupuneri despre segmentul în care este definită variabila rezultat. Ca atare, pentru a o accesa, utilizăm perechea de registre ES:BX, încărcată cu adresa parametrului, preluată din stivă. Aici apare necesitatea ca datele de orice fel (în cazul de față adresa FAR a rezultatului) să fie reprezentate în stivă în același fel ca în memoria de date. Dacă nu am fi respectat convenția de reprezentare a adreselor FAR (offset-ul la adrese mici), instrucțiunea LES BX nu ar fi încărcat corect adresa în perechea de registre ES:BX. Accesul la variabila rezultat este ilustrat în Figura 6.2.

În toate exemplele de mai sus, descărcarea stivei a fost făcută de către programul apelant, printr-o instrucțiune ADD SP. Este posibilă și varianta în care descărcarea stivei se face de către procedură. Acest lucru se implementează printr-o instrucțiune return de forma:

```
ret N
```

unde N este numărul octeților care au fost puși pe stivă ca parametri. Evident, în acest caz nu se mai scrie instrucțiunea ADD SP în programul apelant.

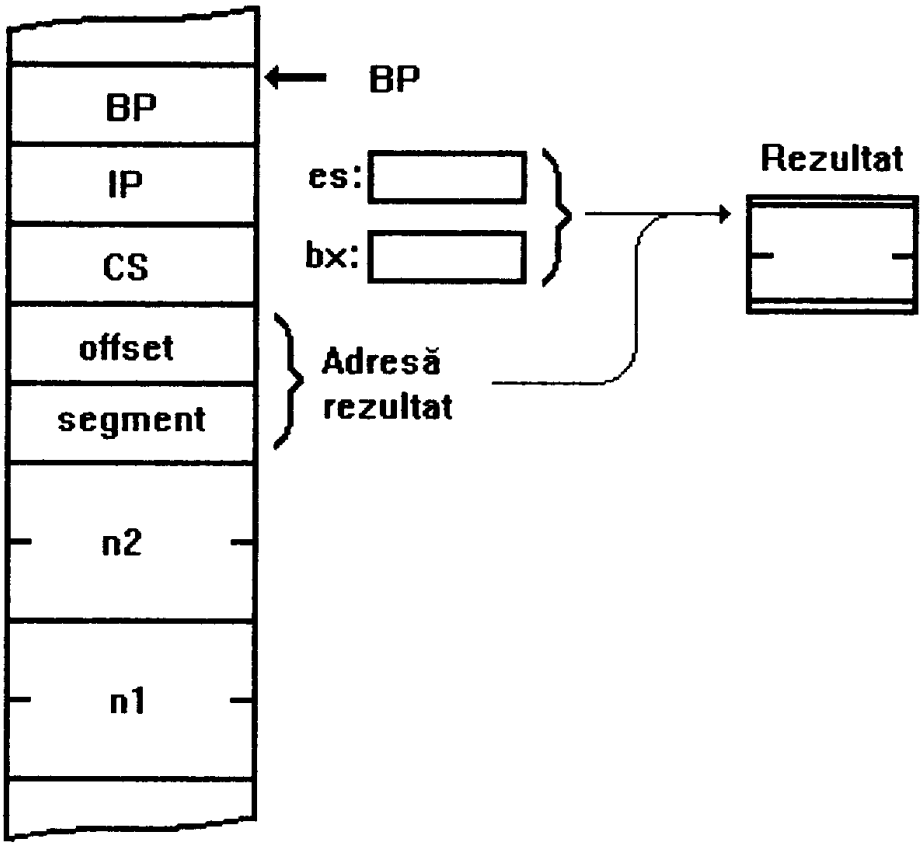


Figura 6.2 Transmiterea unei adrese FAR ca parametru

Problematica transferului parametrilor prin stivă trebuie cunoscută în amănunt atunci când interfațăm module ASM cu module scrise în limbaje de nivel înalt. Este posibil ca diverse proprietăți să difere de la limbaj la limbaj sau chiar de la compilator la compilator.

Spre exemplu, compilatoarele Borland C utilizează următoarea tehnică de transfer a parametrilor:

- parametrii sunt evaluați și plasați în stivă în ordinea inversă a listei de argumente a funcției, adică primul parametru din listă este în vârful stivei, imediat după adresa de revenire;
- stiva este descărcată de programul apelant.

În schimb, compilatoarele Borland Pascal lucrează exact pe dos:

- parametrii sunt plasați în stivă în ordinea din lista de argumente a procedurii (funcției), adică ultimul parametru din listă este în vârful stivei, imediat după adresa de revenire;
- stiva este descărcată de programul apelant.

Imaginea stivei la intrarea într-o procedură C, respectiv Pascal de forma:

```
void f_C (int par_a, int par_b, int par_c);
procedure f_Pascal (int par_a, par_b, par_c)
```

este ilustrată în Figura 6.3.

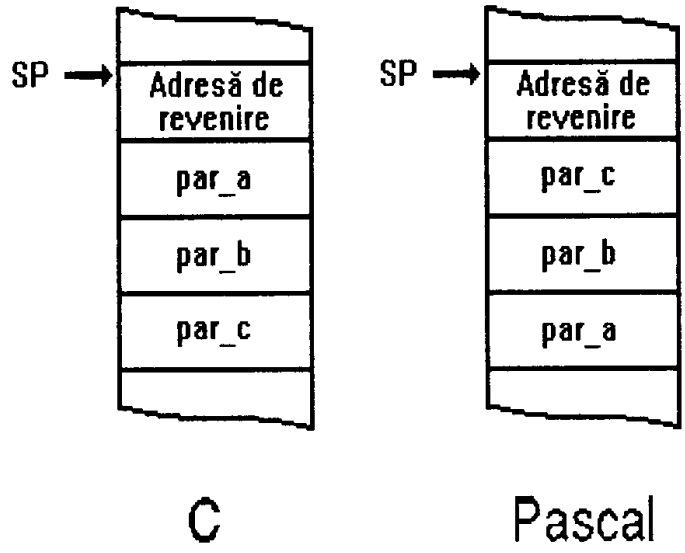


Figura 6.3 Transferul parametrilor în C și Pascal

Secvențele de apel ale celor două proceduri s-ar scrie:

```
push par_c   ;
push par_b   ; Secventa de apel
push par_a   ;
call f_C     ; pentru o functie C
add sp, 6    ;
```

respectiv:

```
push par_a          ;
push par_b          ; Secventa de apel
push par_c          ; pentru o procedura
call f_Pascal       ; (functie) Pascal
```

6.5 Întoarcerea datelor de către proceduri

Procedurile care întorc valori corespund funcțiilor din Pascal sau funcțiilor cu tip nevid din C. În limbaj de asamblare, ne punem problema în sens mai larg, anume ce modalități există pentru a furniza un rezultat programului apelant (inclusiv prin tehnici neortodoxe). Aceste modalități sunt:

a) în lista de parametri apar adresele rezultatelor sau adresa unei zone de date care conține câmpuri pentru rezultate;

b) rezultatele se întorc prin registre;

c) rezultatele se întorc în vârful stivei.

Tehnica a) a fost deja descrisă la transmiterea parametrilor prin zonă de date sau prin stivă. Practic, în interiorul procedurii se depun explicit rezultatele la adresele conținute în parametrii formali respectivi.

Deși această tehnică nu este recomandată ca model în programarea structurată, ea nu se poate evita atunci când rezultatele au dimensiuni mari și sunt prin natura lor gestionate prin adrese. Un exemplu clar este cel al șirurilor de caractere sau al tablourilor în general.

Mai mult decât atât, compilatoarele de nivel înalt utilizează această tehnică (în mod transparent pentru utilizator) atunci când trebuie întoarse tipuri de volum mare prin numele funcției. Concret, se transmite către funcție adresa unei zone temporare de date (ca parametru suplimentar al funcției), în care să se depună rezultatul.

Tehnica b) este folosită cel mai frecvent. De obicei, se folosește registrul acumulator, eventual extins (adică AL, AX, respectiv DX:AX, după cum rezultatul este pe 1, 2 sau 4 octeți).

Dezavantajul acestei tehnici este limitarea la 32 de biți a tipului de date returnat. Totuși, compilatoarele Borland o utilizează ca metodă standard de returnare a tipurilor de date de maxim 32 de biți.

Tehnica c) se folosește destul de rar, fiind total nestandard. Constă în plasarea rezultatelor în vârful stivei din momentul revenirii în programul apelant. Aceasta înseamnă că, practic, rezultatele se suprapun în stivă peste parametrii de apel (se descarcă implicit stiva) și chiar peste adresa de revenire, ceea ce face foarte complicată scrierea procedurii.

Vom prezenta totuși această tehnică pentru că este un exemplu de operație complexă asupra stivei. Considerăm ca de obicei procedura pro_add, de tip far. Parametrii se plasează în stivă în ordinea n1, n2. Secvența de apel este:

```
push word ptr n1+2
push word ptr n1
push word ptr n2+2
push word ptr n2
call far ptr pro_add          ; in acest moment, in
                              ; varful stivei se
                              ; gaseste rezultatul
                              ; adunarii
pop ax                        ; Preia
mov word ptr rez, ax          ; rezultat
pop ax                        ; si descarca
```

```
mov word ptr rez + 2, ax        ; stiva
```

Știm acum ce ar trebui să execute procedura pro_add. Pentru a vedea exact ce acțiuni trebuie implementate, se pornește de la conținutul stivei în momentul intrării în procedură și de la cum trebuie să arate stiva înainte de instrucțiunea de revenire (RET) din procedură. Aceste două situații sunt ilustrate în Figura 6.4.

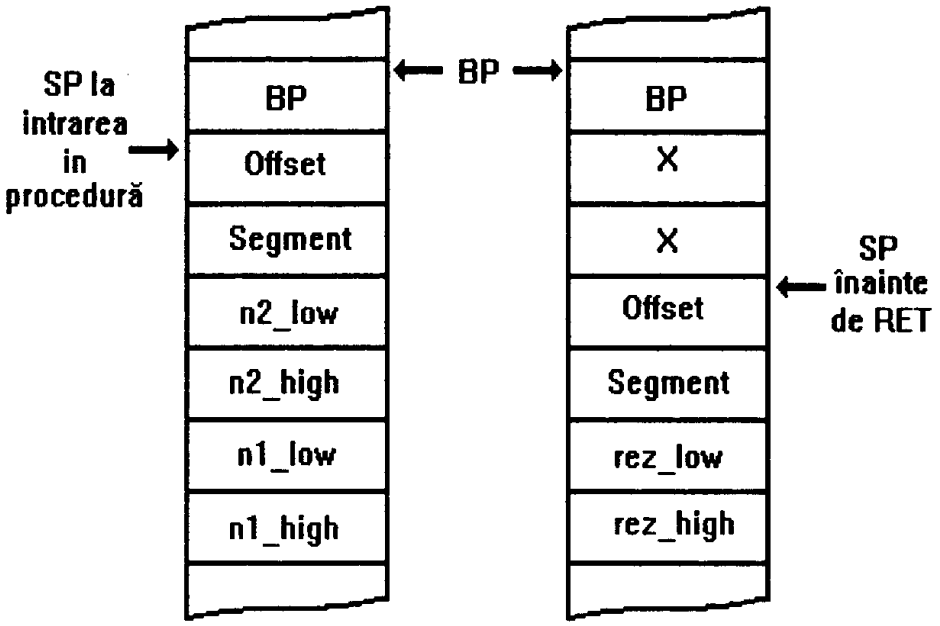


Figura 6.4 Întoarcerea rezultatelor în vârful stivei

Evident, procedura trebuie să construiască imaginea stivei din momentul revenirii în programul apelant, iar secvența de apel trebuie să readucă registrul SP la valoarea inițială.

Pornind de la aceste considerente, definim structuri de acces la stivă conform șablonului de la intrare, respectiv de la ieșire. Pentru șablonul de intrare utilizăm structura sablon_3:

```
sablon_3 struc
        _bp_3       dw ?   ; BP
        _ip_3       dw ?   ; Offset revenire
        _cs_3       dw ?   ; Segment revenire
        n2_low      dw ?   ;
        n2_high     dw ?   ; Parametri
        n1_low      dw ?   ;
        n1_high     dw ?   ;
sablon_3 ends
```

Pentru șablonul de ieșire, definim structura sablon_4:

```
sablon_4 struc
        _bp_4       dw ?
                    dw ?
                    dw ?
        _ip_4       dw ?
        _cs_4       dw ?
        rez_low     dw ?
        rez_high    dw ?
sablon_4 ends
```

Câmpurile _bp_3 și _bp_4 se găsesc în aceeași poziție din memorie. S-au utilizat nume diferite, deoarece sintaxa structurilor impune acest lucru.

Din imaginile stivei din Figura 6.4 rezultă și operațiile care se execută. După calculul sumei, adresa de revenire va trebui deplasată cu 4 octeți în jos în stivă, rezultatul se va depune după noua poziție a adresei de revenire, se va poziționa registrul SP pe adresa de revenire și se va executa RET. Procedura pro_add se implementează astfel:

```
pro_add proc far
        push bp
        mov  bp, sp
        push ax                     ; Salvare
        push dx                     ; registre folosite
        mov  ax, [bp].n1_low        ; Calcul
        add  ax, [bp].n2_low        ; rezultat
        mov  dx, [bp].n1_high
        adc  dx, [bp].n2_high
        mov  [bp].rez_low, ax       ; Depunere rezultat
        mov  [bp].rez_high, dx      ; conform sablonului
                                    ; de iesire
        mov  ax, [bp]._ip_3         ; Pregatire offset adresa
        mov  [bp]._ip_4, ax         ; de revenire
        mov  ax, [bp]._cs_3         ; Pregatire segment adresa
        mov  [bp]._cs_4, ax         ; de revenire
        pop  dx                     ; Refacere
        pop  ax                     ; registre folosite
        pop  bp                     ; Refacere BP
        add  sp, 4                  ; Pozitionare SP pe adresa
                                    ; de revenire
        ret
pro_add enáp
```

În unele situații, este posibil ca zona de stivă în care se depun rezultatele să fie suprapusă peste vechea zonă în care se găsește adresa de revenire. În acest caz, se plasează întâi adresa de revenire în noua poziție din stivă și apoi se depune rezultatul.

6.6 Proceduri cu zone de date proprii (variabile locale)

În multe cazuri, nu putem aloca toate variabilele dintr-o procedură în registrele procesorului. În această situație, procedura trebuie să utilizeze variabile locale proprii, altele decât parametrii formali și decât variabilele alocate în registre. Evident, se pune problema zonelor de memorie în care să fie alocate aceste variabile locale.

Există trei modalități de bază pentru această alocare: în segmentul de date global (.data) vizibil din toate modulele, în stivă sau într-un segment de date propriu.

Prima modalitate, care este evidentă, contrazice, de fapt, caracterul local al acestor variabile și nu se recomandă a fi folosită. Vor fi, deci, detaliate celelalte două metode.

6.6.1 Variabile locale definite în stivă

Ca și tehnica de transmitere a parametrilor prin stivă, această metodă este standardizată, fiind de fapt o extensie a tehnicii de transfer prin stivă. Operațiile care se execută la intrarea în procedură sunt următoarele:

- se salvează BP în stivă;
- se decrementează registrul SP cu numărul necesar de octeți pentru variabilele locale;
- se copiază SP în BP;
- se salvează (eventual) registrele folosite în procedură;
- se accesează parametrii formali și variabilele locale conform șablonului stivei.

În secvența de ieșire din procedură, se execută următoarele operații:

- se refac registrele salvate;
- se incrementează SP cu același număr de octeți cu care a fost incrementat în secvența de intrare;
- se reface registrul BP din stivă;
- se revine în programul apelant (eventual cu descărcarea stivei de parametri.

Pentru exemplificare, să considerăm o procedură pro_local, de tip NEAR care are doi parametri formali de tip WORD și trei variabile locale, toate de tip word. Presupunem că stiva este descărcată de parametrii formali de către procedură. Conform operațiilor de mai sus, definim o structură șablon pentru accesul în stivă:

```
sablon struc
      loc_1 dw ?
      loc_2 dw ?
```

```
        loc_3 dw ?
        _bp   dw ?
        _ip   dw ?
        par_2 dw ?
        par_1 dw ?
sablon ends
```

Schema de dezvoltare a procedurii pro_local este:

```
pro_local proc near
        push bp             ; Salvare BP
        sub  sp, 6          ; Spatiu pentru variabile locale
        mov  bp, sp         ; Acces prin BP
                            ;
                            ; Acces la parametrii formali prin
                            ; expresiile [bp]. par_1,
                            ; [bp].par_2
                            ;
                            ; Acces la variabilele locale prin
                            ; expresiile [bp].loc_1,
                            ; [bp].loc_2, [bp].loc_3
                            ;
        add  sp, 6          ; Refacere spatiu local
        pop  bp             ; Refacere BP
        ret  4              ; Revenire cu descarcare
pro_local endp
```

Imaginea stivei după secvența de intrare în procedură este ilustrată în Figura 6.5.

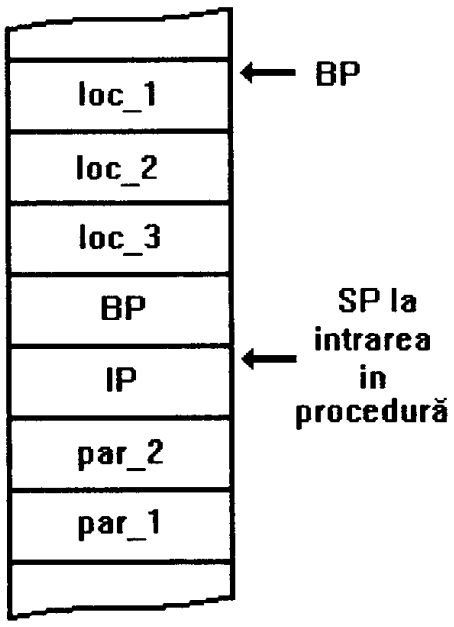


Figura 6.5 Variabile locale alocate în stivă

Variabilele alocate în stivă au unele caracteristici care derivă din metoda de alocare:

- spațiul de memorie alocat nu există decât pe durata procedurii;
- variabilele nu au alocate adrese fixe de memorie: adresele depind de poziția curentă a stivei;
- variabilele nu-și păstrează valoarea de la un apel la altul.

Cea de-a doua proprietate este mai importantă, deoarece există situații în care se dorește în mod explicit ca o variabilă locală să-și păstreze valoarea de la un apel la altul al procedurii. În acest caz, e obligatorie alocarea în segmente proprii de date, deci la adrese fixe.

6.6.2 Variabile locale alocate static

Alocarea variabilelor în segmente proprii de date se mai numește și alocare statică, deoarece variabilele au asociate adrese fixe de memorie.

Ca mod practic de implementare, se definește un segment de date local cu ajutorul directivei SEGMENT, ceea ce previne gruparea acestui segment cu alte segmente. Accesul la segmentul local se va realiza prin unul din registrele DS sau ES, urmând ca celălalt registru de segment de date să fie utilizat (dacă este cazul) pentru accesul la segmentul de date al programului apelant.

Să considerăm aceeași procedură pro_local, de data aceasta de tip FAR, cu trei variabile locale și doi parametri formali. Definim segmentul local prin:

```
local_data segment
        loc_1 dw ?
        loc_2 dw ?
        loc_3 dw ?
local_data ends
```

și convenim să accesăm acest segment prin registrul ES. Ca atare, definiția procedurii pro_local va fi încadrată de directive ASSUME corespunzătoare.

Parametrii formali sunt transmiși prin stivă, conform șablonului de acces:

```
sablon struc
        _bp     dw ?
        _cs_ip  dd ?
        par_2   dw ?
        par_1   dw ?
sablon ends

ASSUME es:local_data
pro_local proc far
        push bp
        mov  bp, sp
        push es
        mov  es, SEG local_data
                                    ;
                                    ; DS : asa cum vine
```

```
                                    ; din programul apelant
                                    ; ES : pozitionat
                                    ; pe segmentul local
                                    ;
                                    ; Acces la parametrii
                                    ; formali prin
                                    ; expresiile [bp]. par_1,
                                    ; [bp].par_2
                                    ;
                                    ; Acces la variabilele
                                    ; locale prin
                                    ; expresiile loc_1,
                                    ; loc_2, loc_3
                                    ; sau, mai clar,  es:loc_1,
                                    ; es:loc_2, es:loc_3
                                    ;
            pop es
            pop bp
            ret 4
      pro_local endp
ASSUME es:nothing
```

Există situația în care unul sau mai mulți parametri formali sunt pointeri (adrese) far. În acest caz, se salvează unul din registrele DS și ES în stivă și se încarcă pointerul respectiv într-o pereche adecvată de registre (cu instrucțiunile LDS sau LES).

Să presupunem, de exemplu, că unul din parametrii formali, descris în șablonul de acces prin par_far, indică un cuvânt care trebuie copiat în variabila locală loc_1. Secvența de copiere va fi:

```
push ds                    ; Salvare
lds bx, [bp].par_far       ; incarcare pointer
mov ax, [bx]               ; Acces
mov es:loc_1, ax           ; Copiere in segment local
pop ds
```

Dacă unul din parametri este o adresă far de procedură, se poate executa direct un apel indirect, prin:

```
call dword ptr [bp].par_far
```

Să considerăm un exemplu în care utilizarea variabilelor locale statice este obligatorie, anume un generator simplu de numere aleatoare pe 16 biți, implementat prin procedura de tip FAR rand_asm. Procedura are ca parametru un întreg pe 16 biți fără semn (notat generic n) și întoarce în AX un număr aleator fără semn în domeniul 0...n-1.

Metoda de generare se bazează pe o variabilă locală X, asupra căreia se execută operația:

X ← partea mediană (X•X + C)

unde X•X este pătratul lui X (pe 32 de biți), iar C este o constantă pe 32 de biți. Prin parte mediană se înțeleg biții 8...23 din cei 32 de biți ai expresiei calculate (vezi Figura 6.6).

După ce s-a calculat noua valoare a lui X, se întoarce programului apelant valoarea X MOD n, deci un număr între 0 și n-1. Toți operanzii se consideră fără semn.

În astfel de operații, trebuie analizată cu atenție problema depășirilor. Calculul X mod n se face printr-o împărțire a lui X la n, unde atât X, cât și n sunt pe 16 biți. Se va executa de fapt o împărțire DX:AX la n, cu DX = 0, fapt care asigură evitarea depășirii: cea mai mare valoare posibilă a lui X (65535), împărțită la cea mai mică valoare nebanală a lui n (2) nu produce depășire.

Caracterul pseudo-aleator al algoritmului de mai sus provine în mod esențial din faptul că variabila X își păstrează valoarea de la un apel la altul al procedurii (vechea valoare a lui X participă la calculul noii valori).

Pentru accesul la parametrul n, se utilizează o structură șablon.

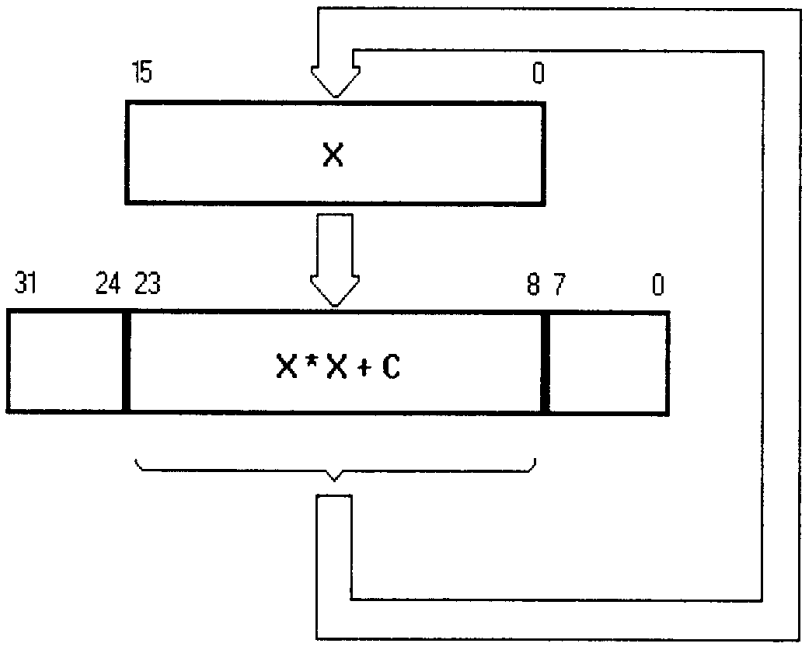


Figura 6.6 Un generator simplu de numere aleatoare

Conform metodei de dezvoltare expuse mai sus, definim un segment local în care rezervăm spațiu pentru variabila de tip word X și pentru constanta de tip double-word C (ambele inițializate cu câte o valoare oarecare).

Deoarece va trebui să executăm o adunare pe 32 de biți, avem nevoie de acces explicit la părțile low și high ale constantei C, păstrând definiția ei ca variabilă

double-word. Acest lucru este realizat prin operatorul THIS, cu ajutorul căruia se definesc constantele simbolice c_lo și c_hi.

Calculele vor folosi registrul AX, respectiv perechea de registre DX:AX, care este operand implicit la înmulțiri și împărțiri. Implementarea este următoarea (presupunem un fișier sursă cu numele rand.asm):

```
;
; Fisier RAND.ASM
;
.model large
public rand_asm, init_asm

sablon struc
      _bp     dw ?          ; sablon de
      _cs_ip  dd ?          ; de acces
      n       dw ?          ; la stiva
sablon ends

local_data segment
x     dw 111                ; Variabila locala statica X
c_lo  equ this word
c_hi  equ c_lo + 2
c     dd 115321             ; Constanta C
local_data ends

.code
assume es:local_data
rand_asm proc far
push bp                     ; Secventa standard
mov  bp, sp                 ; de intrare
                            ;
pushf                       ; Salvare bistabili
cmp [bp].n, 2               ; Test caz banal (n = 0,1)
mov ax, [bp].n              ; Se intoarce chiar n
jb gata
push es                     ; Salvari registre
push dx                     ; folosite
                            ;
mov ax, SEG local_data      ; Pozitionare ES
mov es, ax                  ; pe segmentul local
                            ;
mov ax, es:x                ; Variabila X
mul es:x                    ; Ridicare la patrat (DX:AX)
add ax, c_lo                ; Adunare
adc dx, c_hi                ; cu C
mov al, ah                  ; Luarea partii mediane
mov ah, dl                  ; din DX:AX in AX
mov es:x, ax                ; Depunere in X
xor dx, dx                  ; Pregatire impartire
div [bp].n                  ; impartire (0:X) la n
mov ax, dx                  ; Catul se pune in AX
                            ;
pop dx                      ; Refaceri
pop es                      ; registre
```

```
gata:
        popf                            ; Refacere bistabili
                                        ;
        pop bp                          ; Secventa standard
        ret                             ; de iesire
        rand_asm endp
                                        ;
                                        ; Initializare generator
                                        ;
        init_rand proc far
        push bp
        mov bp, sp
        push es                         ; Salvari
        push ax
        mov ax, SEG local_data
        mov es, ax
        mov ax, [bp].n                  ; Copiere parametru
        mov es:x, ax                    ; in variabila X
        pop ax
        pop es                          ; Refaceri
        pop bp
        ret
        init_rand endp
        end
```

Procedura rand_asm este completată cu o rutină de inițializare a generatorului, având un parametru de tip word transmis prin stivă, care inițializează variabila X. Se folosește aceeași structură șablon pentru accesul la stivă.

Să considerăm un exemplu de apel al acestor proceduri, pe care le presupunem definite în fișierul sursă rand.asm. Dorim inițializarea cu valori aleatoare în gama 0...9999 a unui tablou de 256 de întregi, pe care îl afișăm, îl sortăm crescător și apoi îl afișăm din nou. Pentru afișare și sortare utilizăm procedurile tipvec și bubble, dezvoltate în 2.7, pe care le presupunem definite într-un fișier sursă cu numele bubble.asm, care conține declarații PUBLIC ale procedurilor în cauză.

În programul principal (presupus în fișierul sursă main.asm), cele patru proceduri se declară ca simboluri externe și se folosesc conform metodei proprii de transfer al parametrilor:

```
;
; Fisier MAIN.ASM
;
.model large
include io.h
extrn rand_asm: far, init_rand: far, tipvec: far, bubble: far
.stack 1024
.data
        vector dw 256 dup (?)
        dim    dw ($-vector)/2
.code
start:
        init_ds_es
```

```
        mov  ax, 1131           ; Valoare initiala
        push ax                 ; a generatorului
        call init_rand
        add  sp, 2              ; Descarcare stiva
;
        mov  cx, dim            ; Dimensiune vector
        lea  bx, vector         ; Adresa vector
reluare:
        mov  ax, 10000          ; Domeniu 0...9999
        push ax
        call rand_asm
        add  sp, 2              ; Descarcare stiva
        mov  [bx], ax           ; Depunere valoare
        add  bx, 2              ; Actualizare adresa
        loop reluare
;
        putsi <'Vector nesortat', cr, lf>
        lea  si, vector         ; Adresa tablou
        mov  cx, dim            ; Numar de elemente
        call tipvec             ; Afisare tablou nesortat
;
        lea  bx, vector         ; Adresa tablou
        mov  cx, dim            ; Numar de elemente
        call bubble             ; Sortare tablou
;
        putsi <cr,lf,'Vector sortat', cr, lf>
        lea  si, vector
        mov  cx, dim
        call tipvec             ; Afisare tablou sortat
;
      exit_dos                  ; Iesire in DOS
      end start
```

Secvența de dezvoltare a acestei aplicații este:

```
> tasm rand.asm
> tasm bubble.asm
> tasm main.asm
> tlink main rand bubble io, main
```

în urma căreia rezultă un fișier executabil main.exe.

Exemplul de mai sus constituie și un model de dezvoltare modulară a unei aplicații. Se observă că o proiectare îngrijită a modulelor permite refolosirea lor în diverse contexte.

6.7 Proceduri recursive

În limbaj de asamblare, nu există restricții asupra recursivității: orice procedură proc_a se poate apela pe ea însăși (recursivitate directă) sau poate apela o procedură proc_b care, la rândul ei apelează procedura proc_a (recursivitate indirectă).

În dezvoltarea unei proceduri recursive, trebuie pornit de la specificarea algoritmului recursiv. Specificarea unui asemenea algoritm pune în evidență un caz de bază (care oprește recursivitatea) și un caz general, în care se invocă același algoritm, aplicat altui set de date. Specificarea corectă a cazului de bază și garantarea faptului că acesta este atins întotdeauna, indiferent de setul de date de intrare primit, sunt punctele-cheie ale proiectării unui algoritm recursiv.

Să considerăm, ca exemplu, o procedură de afișare în baza 10 a unui număr pe 16 biți fără semn. La nivelul funcțiilor de intrare/ieșire, avem posibilitatea de a afișa caractere ASCII. Dorim deci un algoritm recursiv care să genereze cifrele zecimale ale numărului, în ordinea afișării. În 2.2.5 a fost dat un algoritm nerecursiv, care producea cifrele zecimale ale numărului în ordine inversă.

Algoritmul recursiv se specifică în felul următor:

```
putu_proc (n) {
        daca (n < 10)
        afiseaza (n + '0')
        altfel {
                putu_proc (n/10)
                afiseaza (n MOD 10 + '0')
        }
}
```

Cazul de bază este (n < 10), pentru care se face doar afișarea cifrei și revenirea în programul apelant. În Figura 6.7 sunt ilustrate apelurile recursive în cazul afișării numărului 123 și valorile succesive ale parametrului n.

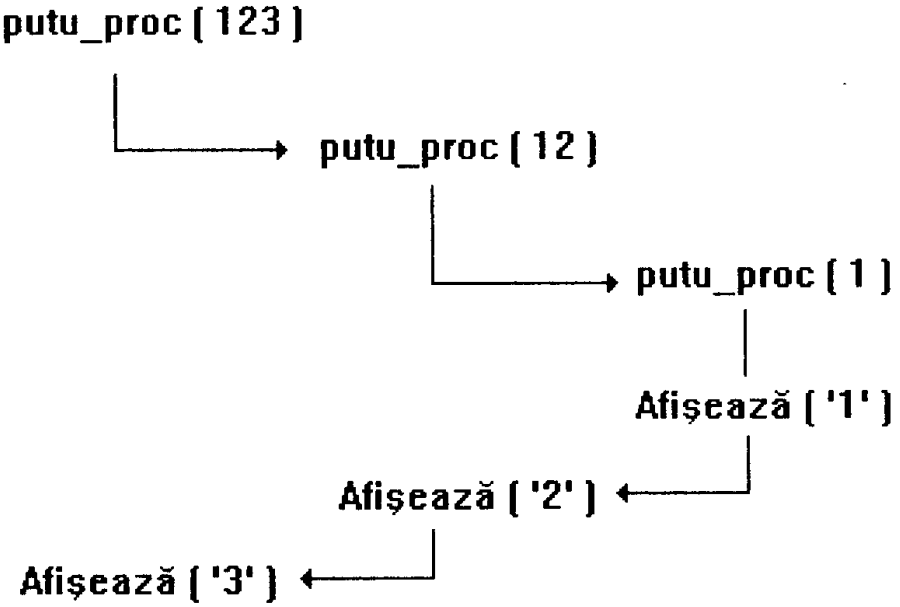


Figura 6.7 Apelul recursiv al procedurii putu_proc

Regulile generale de implementare a procedurilor recursive sunt:

- fiecare apel al procedurii nu trebuie să afecteze datele apelurilor precedente (procedura să fie reentrantă);
- nu se folosesc variabile locale alocate static, ci numai în stivă sau în registre;
- parametrii se transmit prin stivă (se asigură implicit faptul că parametrii sunt locali fiecărui apel);
- registrele ale căror valori au semnificație de la un apel la altul se salvează în stivă;
- dacă procedura întoarce valori prin registre, rezultatele intermediare trebuie memorate în stivă.

Pentru implementarea procedurii putu_proc, se va utiliza funcția DOS 2, care afișează la consolă caracterul primit în registrul DL. Pentru accesul la stivă, considerăm structura șablon:

```
sablon struc
      _bp dw ?
      _cs_ip dw ?
      n dw ?
sablon ends
```

Pentru o implementare mai eficientă, rescriem algoritmul recursiv în forma:

```
putu_proc (n) {
      daca (n < 10)
            x ← n
      altfel {
            putu_proc (n/10)
            x ← n MOD 10
            }
      afiseaza x + '0'
}
```

Alocarea variabilelor este următoarea:

- n - în stivă, accesibil prin expresia [bp].n
- x - în registrul DL (care se salvează/restaurează)

Implementarea este următoarea:

```
putu_proc proc far
      push bp                  ; Secventa
      mov  bp,sp               ; standard
      push dx                  ; Salvari
      push ax                  ; registre
      push bx                  ; folosite
                               ;
      mov ax, [bp].n           ; Test caz
      cmp ax, 10               ; de baza
      mov dl, al               ; x <-- n
      jb  p_u_1                ; Salt la afisare
```

```
                                    ;
        mov  bx, 10                 ; Caz general
        xor  dx, dx                 ; Se calculeaza
        div  bx                     ; n/10 si n MOD 10
                                    ; AX = n/10
                                    ; DX (DL) = x = n MOD 10
        push ax                     ; Apel recursiv cu
        call far ptr putu_proc      ; n/10 ca parametru
p_u_1:
        add  dl, '0'                ; x + '0'
        mov  ah, 2                  ; Cod functie DOS
        int  21H                    ; Afisare


         pop bx                     ; Refaceri
         pop ax                     ; registre
         pop dx
         pop bp
         retf 2                     ; Return cu descarcare
putu_proc endp
```

În implementarea de mai sus, este esențială integritatea variabilei x de la un apel la altul, deci salvarea/restaurarea registrului DX. Pentru a preveni depășirile, se face împărțirea 32 de biți la 16 biți, deci constanta 10 se încarcă într-un registru de 16 biți. După împărțire, DX (de fapt DL) conține restul n MOD 10, iar AX câtul (n/10). Se pune, deci, AX în stivă și se apelează recursiv procedura. După revenire, se afișează cifra din DL.

Putem acum dezvolta o procedură care să afișeze un număr pe 16 biți cu semn, după algoritmul:

```
puti_proc (n) {
        daca (n < 0) {
              n ← - n
              afiseaza ('-')
        }
        putu_proc (n)
}
```

Dacă numărul e negativ, se complementează față de 2 și se afișează semnul '-', după care se afișează numărul complementat. Implementarea este următoarea:

```
puti_proc proc far
        push bp                     ; Secventa
        mov  bp, sp                 ; standard
        push ax                     ; Salvari
        push dx                     ; registre
                                    ;
        mov  ax, [bp].n             ; Parametrul n
        or   ax, ax                 ; Pozitionare bistabili
        jns  p_i_1                  ; Test bit de semn
        mov  dl, '-'                ; Afisare
        mov  ah, 2                  ; semn '-'
```

```
        int  21H
        neg  word ptr [bp].n     ; Complementare n
p_i_1:
        push [bp].n              ; Pregatire parametru
        call far ptr putu_proc   ; Apel putu_proc
                                 ;
        pop  dx                  ; Refaceri
        pop  ax                  ; registre
        pop  bp
        retf 2                   ; Return cu descarcare
puti_proc endp
```

Se observă forma directă de plasare a parametrului [bp].n în stivă, pentru apelul procedurii putu_proc.

Putem scrie acum și macroinstrucțiuni de apel adecvate:

```
puti macro X
        push X
        call far ptr puti_proc
endm
putu macro X
        push X
        call far ptr putu_proc
endm
```

După modelul procedurii putu_proc, putem scrie o procedură care să afișeze la consolă reprezentarea unui număr pe 16 biți fără semn într-o bază de numerație dată. Baza se consideră în domeniul 2...36, iar cifrele într-o bază mai mare ca 10 sunt '0'...'9', 'A', 'B' etc. Limitarea bazei la valoarea 36 derivă din această alegere a cifrelor.

Algoritmul este următorul:

```
put_base (n, baza) {
        daca (baza < 2 sau baza > 36)
                return
        daca (n < baza)
                x ← n
        altfel {
                put_base (n/baza, baza)
                x ← n MOD baza
        }
        daca (x < 10)
                afiseaza (x + '0')
        altfel
                afiseaza (x + 'A' - 10)
}
```

Se începe printr-un test de corectitudine a bazei: dacă aceasta nu este în domeniul admisibil, se revine imediat în programul apelant. Se execută apoi același algoritm recursiv ca în procedura putu_proc, înlocuind constanta 10 cu

variabila baza. Afișarea cifrei curente x se face diferențiat: dacă este mai mică sau egală cu 10, se afișează ca cifră zecimală; altfel, se afișează ca literă.

Pentru accesul la stivă, se utilizează structura șablon de mai jos:

```
sablon      struc
            _bp     dw ?
            _cs_ip  dd ?
            baza    dw ?
            n       dw ?
sablon      ends
```

Implementarea este următoarea (se consideră că stiva este descărcată de programul apelant):

```
put_base proc far
      push  bp                          ; Secventa standard
      mov   bp, sp
      cmp   word ptr [bp].baza, 2       ; Test baza corecta
      jb    err_exit                    ; baza < 2
      cmp   word ptr [bp].baza, 36
      ja    err_exit                    ; baza > 36
;
      push  ax                          ; Salvari
      push  dx                          ; registre
      mov   ax, [bp].n                  ; Test
      cmp   ax, [bp].baza               ; n < baza ?
      mov   dl, al                      ; x <-- n
      jb    p_b_1                       ; Salt la afisare
      xor   dx, dx                      ; Calcul n/baza
      div   [bp].baza                   ; si n MOD baza
                                        ; AX = n/baza
                                        ; DX (DL) = n MOD baza
;
;     Pregatire parametri pentru apelul recursiv
;
      push  ax                          ; n/baza
      push  [bp].baza                   ; baza
      call  far ptr put_base            ; Apel recursiv
      add   sp, 4                       ; Descarcare stiva
p_b_1:
      mov   ah, 2                       ; Cod functie DOS
      cmp   dl, 10                      ; x < 10 ?
      jae   p_b_2
      add   dl, '0'                     ; x + '0'
      jmp   p_b_3
p_b_2:
      add   dl, 'A' - 10                ; x + 'A' - 10
p_b_3:
      int   21H                         ; Afisare
      pop   dx                          ; Refaceri
      pop   ax                          ; registre
err_exit:
      pop   bp
      ret                               ; Revenire
```

```
put_base endp
```

Deoarece toate variabilele implicate sunt fără semn, se folosesc salturi condiționate de „below" și „above".

Un mic program de test (listat în continuare) citește un număr și o bază de la consolă și afișează numărul în baza dată, după care reia procesul de citire. Dacă numărul introdus este 0, programul se termină.

```
.model large
      include io.h
.stack 1024
.code
start:
      init_ds_es
reluare:
      putsi <cr,lf,'Numar = '>      ; Prompter
      getu                          ; Citire intreg
      or    ax,ax                   ; Test oprire ?
      jz    exit                    ; Salt la iesire
      push  ax                      ; Plasare n in stiva
      putsi <'Baza = '>             ; Prompter
      getu                          ; Citire intreg
      push  ax                      ; Plasare baza in stiva
      call  far ptr put_base        ; Apel
      add   sp, 4                   ; Descarcare stiva
      jmp   reluare                 ; Reluare
exit:
      exit_dos
      end   start
```

Un caz deosebit îl constituie funcțiile recursive, adică procedurile care întorc valori. Trebuie acordată atenție memorării valorilor întoarse de apelurile recursive. Să considerăm o funcție recursivă care întoarce numărul Fibonacci de ordinul n, definită recursiv prin:

```
fib (0) = fib (1) = 1
fib (n) = fib (n-1) + fib (n-2), pentru n >= 2
```

Această funcție a fost evaluată prin iterație în 2.4.4. Vom studia acum implementarea recursivă.

Se observă că, pentru valoarea lui fib(n), sunt necesare două apeluri recursive și salvarea rezultatului întors de primul apel. Presupunem că parametrul n se transmite prin stivă, că procedura este de tip FAR, că întoarce rezultatul în AX (ca număr fără semn) și că stiva este descărcată de programul apelant. Șablonul de acces este definit prin structura:

```
sablon struc
       _bp     dw ?
       _cs_ip  dd ?
      n        dw ?
sablon ends
```

iar implementarea este următoarea:

```
fib proc far
;
;       Primeste in stiva numarul n
;       Intoarce în AX valoarea fib (n)
;
        push    bp
        mov     bp, sp
        push    bx              ; Salvare registru folosit
;
        mov     ax, [bp].n      ; Preia argumentul n
        cmp     ax, 1           ; Cazul de baza: n <= 1
        jbe     fib_1           ; Salt la evaluare imediata
;
        dec     ax              ; Pregatire apel
        push    ax              ; pentru n-1
        call    fib             ; AX = fib (n-1)
        add     sp, 2           ; Descarcare stiva
        mov     bx, ax          ; Rezultat intermediar
;
        mov     ax, [bp].n      ; Pregatire
        dec     ax              ; apel
        dec     ax              ; pentru
        push    ax              ; n-2
        call    fib             ; AX = fib (n-2)
        add     sp,2            ; Descarcare stiva
;
; in acest moment, AX = fib (n-2), BX = fib (n-1)
;
        add     ax, bx          ; Calcul fib (n)
        jmp     fib_2
fib_1:
        mov     ax, 1           ; Evaluare fib (0) si fib (1)
fib_2:
        pop     bx              ; Restaurare
        pop     bp
        ret                     ; Revenire
fib endp
```

În implementarea de mai sus, este esențială salvarea și restaurarea registrului BX, care memorează rezultatul intermediar fib (n-1). Programul de test al acestei proceduri este lăsat ca exercițiu pentru cititor.

Nu putem încheia discuția despre proceduri recursive fără a vorbi despre problema clasică a turnurilor din Hanoi. În această problemă, se consideră n discuri de diametre distincte, așezate pe un suport vertical (turn sursă), în ordinea descrescătoare a diametrelor (vezi Figura 6.8). Se cere să se mute întreg ansamblul pe un alt turn (destinație), folosind un turn de manevră. Discurile se pot muta unul câte unul, cu restricția că un disc nu se poate așeza decât deasupra unui disc de diametru mai mare.

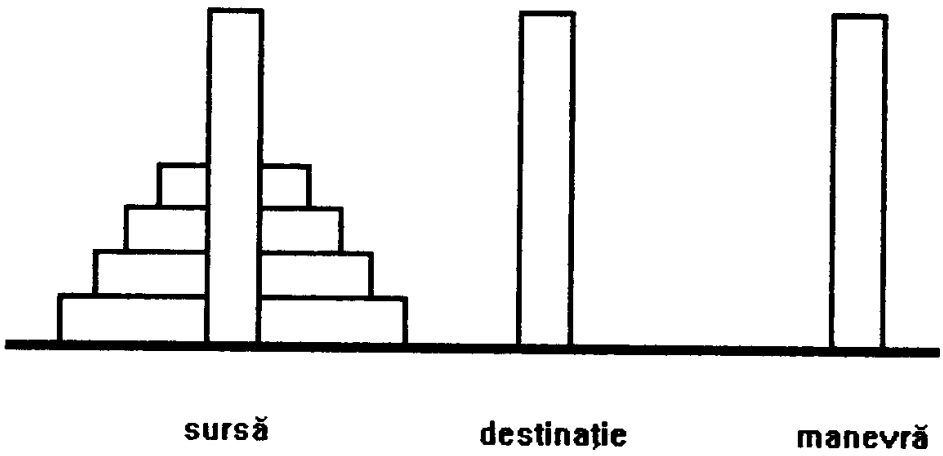


Figura 6.8 Turnurile din Hanoi

Algoritmul care rezolvă această problemă este în mod natural recursiv și se bazează pe observația că, dacă n = 1, adică există un singur disc, acesta se poate muta direct pe turnul destinație. Dacă notăm simbolic operația de mutare prin:

```
move (n, sursa, destinatie, manevra)
```

unde sursă, destinație și manevră sunt cele trei turnuri, algoritmul se implementează prin:

```
move (n, sursa, destinatie, manevra) {
      daca (n = 1)
            Muta discul de pe sursa pe destinatie
      altfel {
            move (n-1, sursa, manevra, destinatie)
            move (1, sursa, destinatie, manevra)
            move (n-1, manevra, destinatie, sursa)
      }
}
```

Se mută primele n-1 discuri de pe turnul sursă pe turnul manevră, folosind ca spațiu de lucru turnul destinație. În acest moment, turnul sursă conține un singur disc, turnul destinație este liber, iar turnul manevră conține n-1 discuri. Se mută unicul disc de pe turnul sursă pe turnul destinație. În acest moment, turnul sursă este liber, turnul destinație conține discul cu diametrul cel mai mare, iar turnul manevră conține celelalte n-1 discuri. Se aplică recursiv algoritmul, mutând n-1 discuri de pe turnul manevră pe turnul destinație și folosind turnul sursă ca spațiu de lucru.

Pentru un n dat, se deplasează explicit discul cu diametrul cel mai mare, ceea ce asigură îndeplinirea restricției din enunț.

Implementarea algoritmului se va face prin mesaje explicite la consolă, turnurile fiind codificate prin caractere.

Implementarea algoritmului, împreună cu un program de test, este listată în continuare. Structura șablon definește ordinea de plasare a parametrilor în stivă: n, sursă, destinație, manevră, toate de tip word. Toate apelurile recursive vor utiliza această ordine.

Simularea mutărilor se face prin afișarea unui mesaj, definit în zona de date. Câmpurile sursa_asc și dest_asc sunt variabile și vor identifica (prin câte un caracter) turnurile respective. Când n = 1 (cazul de bază), se depun caracterele care identifică turnurile în șirul de caractere mesaj și se afișează acest șir. Implementarea apelurilor recursive constă în plasarea în stivă a parametrilor în ordinea precizată și apelul procedurii.

Programul principal citește ciclic de la consolă numărul n, afișează un mesaj de identificare, pregătește parametrii în stivă și apelează procedura move. Bucla se oprește când se introduce valoarea 0 pentru n.

```
.model large
include io.h
.stack        4096
      sablon struc
            _bp      dw ?
            _cs_ip   dw 2 dup (?)
            manevra  dw ?
            dest     dw ?
            sursa    dw ?
            n        dw ?
      sablon ends
.data
      mesaj          db cr, lf, 'Muta discul de pe turnul '
      sursa_asc      db ?
                     db ' pe turnul '
      dest_asc       db ?
                     db 0
.code
      move proc far
;
; Muta n discuri de pe sursa pe dest
; folosind manevra ca zona de lucru
;
      push  bp
      mov   bp, sp
      cmp   word ptr [bp].n, 1; Test caz de baza (n = 1)
      jne   move_1             ; Salt la cazul general
;
; Mutare explicita
;
      mov   bx, [bp].sursa     ; Pregatire mesaj
      mov   sursa_asc, bl      ; Disc sursa
      mov   bx,[bp].dest
      mov   dest_asc,bl        ; Disc destinatie
```

```
        puts  mesaj                ; Afisare mesaj
        pop   bp
        retf  8                    ; Revenire cu descarcare
move_1:
;
; Cazul general
;
        mov   ax, [bp].n           ; Pregatire parametri
        dec   ax                   ; n-1
        push  ax
        push  [bp].sursa           ; Sursa
        push  [bp].manevra         ; Actuala manevra devine
                                   ; destinatie
        push  [bp].dest            ; Actuala destinatie
                                   ; devine manevra
        call far ptr move          ; Primul apel recursiv
;
        mov   ax, 1                ; Pregatire parametri
        push  ax                   ; n = 1
        push  [bp].sursa           ; Sursa
        push  [bp].dest            ; Destinatie
        push  [bp].manevra         ; Manevra
        call  far ptr move         ; Al doilea apel recursiv
;
        mov   ax, [bp].n           ; Pregatire parametri
        dec   ax
        push  ax                   ; n-1
        push  [bp].manevra         ; Actuala manevra devine
                                   ; sursa
        push  [bp].dest            ; Destinatie
        push  [bp].sursa           ; Actuala sursa devine
                                   ; manevra
        call  far ptr move         ; Al treilea apel recursiv
;
        pop   bp                   ; Revenire cu
        retf  8                    ; descarcare
        move  endp
;
; Program principal
;
start:
        init_ds_es
reluare:
        putsi <cr,lf,'Numar discuri: '>
        geti
        or    ax, ax
        jz    exit                       ; n = 0, stop
        putsi <cr,lf,'Muta '>
        puti  ax                         ; Afisare n
        putsi <' turnuri, de pe A pe B,'>
        putsi <' folosind C ca'>
        putsi <' manevra', cr, lf>
;
        push  ax                         ; n
        mov   ax, 'A'
```

```
        push  ax                        ; Sursa
        mov   ax, 'B'
        push  ax                        ; Destinatie
        mov   ax, 'C'
        push  ax                        ; Manevra
        call  far ptr move              ; Apel
        jmp   reluare                   ; Reluare
exit:
        exit_dos
        end   start
```

La implementarea algoritmilor recursivi, trebuie făcută o estimare a spațiului necesar de stivă. Spre exemplu, procedura move de mai sus necesită n nivele de apel recursiv. La fiecare nivel, se consumă 12 octeți din stivă (vezi structura șablon). Ca atare, volumul necesar de stivă este de ordinul 12*n. Dacă estimăm că, într-o aplicație dată, valoarea lui n nu va crește peste 100, stiva acelei aplicații trebuie să fie de cel puțin 1200 de octeți.

Înainte de a trece la implementarea unui algoritm recursiv, trebuie să ne convingem (uneori prin demonstrații teoretice complicate), că algoritmul se termină (se atinge cazul de bază) după un număr finit de pași. Nu toate funcțiile definite recursiv au această proprietate. De exemplu, funcția f, „definită" pentru orice n pozitiv prin:

```
f(n) = 5, pentru n = 7
f(n) = f(f(n+2)), pentru n diferit de 7
```

nu este nici măcar o funcție corect definită. Încercarea de a evalua f(5) conduce imediat la o tautologie (f(5) depinde de f(5)), iar pentru n diferit de 5 și de 7 se obține un ciclu infinit.

Trebuie deci acordată atenție descrierii teoretice a algoritmului recursiv, înainte de a trece la implementare.

Observațiile despre implementarea algoritmilor recursivi sunt valabile și în cazul limbajelor de nivel înalt.

6.8 Proceduri cu număr variabil de parametri

Dezvoltarea procedurilor cu număr variabil de parametri are ca scop, pe de o parte, punerea în evidență a flexibilității limbajului de asamblare și, pe de altă parte, ilustrarea modului de implementare a unui mecanism care există în anumite limbaje de nivel înalt (de exemplu în C).

Dezvoltarea unei proceduri cu număr variabil de parametri se bazează pe următoarele reguli:

- parametrii se transmit prin stivă;
- există un număr (cel puțin 1) de parametri ficși (care nu pot lipsi) și care se transmit în vârful stivei (imediat după adresa de revenire);

- unul din parametrii care sunt totdeauna prezenți conține informații (codificate într-o formă oarecare) despre numărul și tipul parametrilor actuali;
- în procedură se analizează parametrii ficși, deducându-se numărul și tipul parametrilor variabili, care sunt apoi extrași din stivă;
- parametrii variabili ca număr transmiși în stivă trebuie să corespundă cu descrierea lor în parametrii ficși;
- stiva este descărcată de către programul apelant.

Regulile de mai sus nu sunt alese întâmplător. Astfel, faptul că parametrii ficși se găsesc în vârful stivei face ca aceștia să fie accesibili prin metoda standard. Programul apelant este cel care știe precis câți octeți a plasat în stivă înainte de apel, deci este natural ca acesta să descarce stiva.

Se înțelege acum de ce, în limbajul C, ordinea plasării argumentelor unei proceduri în stivă este de la dreapta la stânga: în acest fel, la o procedură cu număr variabil de parametri, parametrii ficși (care se află la începutul listei de argumente) vor fi plasați imediat după adresa de revenire.

Imaginea stivei la intrarea în procedură este ilustrată în Figura 6.9.

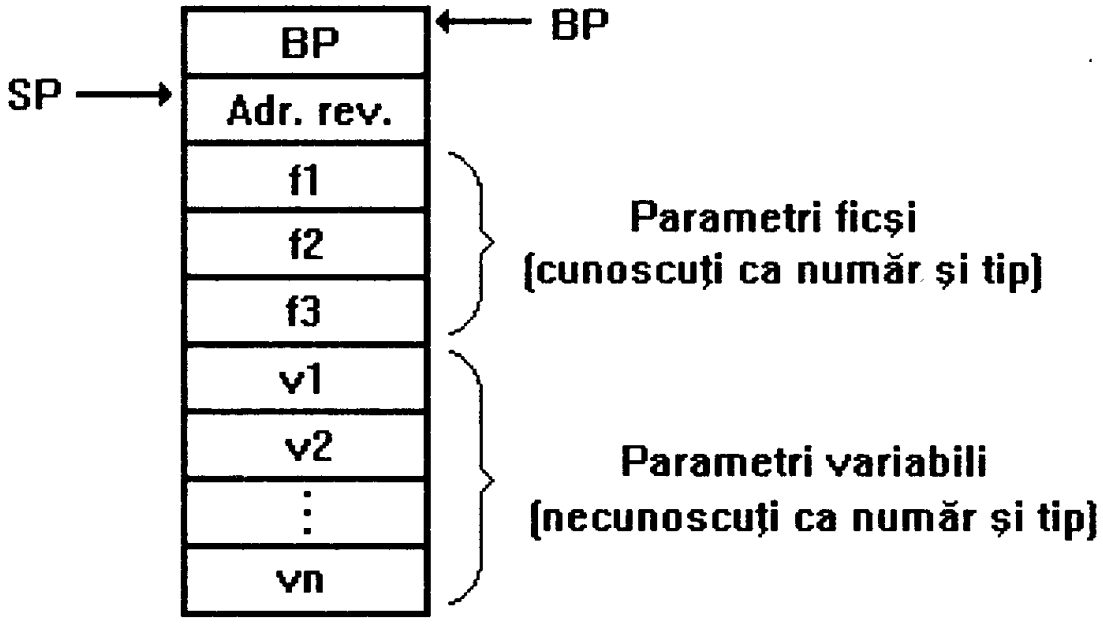


Figura 6.9 Stiva la intrarea într-o procedură cu număr variabil de parametri

Să considerăm un exemplu tipic de procedură cu număr variabil de parametri: o procedură printf_proc de afișare cu format. Există un parametru fix, de tip

adresă NEAR a unui șir de caractere care descrie formatul de afișare. Șirul de caractere poate conține:

- caractere obișnuite, care se afișează ca atare;
- secvențe escape:
- '\n' pentru CR și LF;
- '\\' pentru caracterul '\';
- '%%' pentru caracterul '%';
- descriptori de format, compuși din caracterul % și un al doilea caracter care descrie tipul afișării:
  - %s pentru șiruri de caractere;
  - %d pentru întregi pe 16 biți cu semn;
  - %u pentru întregi pe 16 biți fără semn.

Pentru fiecare specificator de format, se presupune că există un parametru variabil în stivă, care trebuie afișat corespunzător. Pentru întregi, se transmite în stivă conținutul care trebuie afișat, iar pentru șiruri, se transmite adresa de început, de tip NEAR. Se observă că, în toate situațiile, se transmite în stivă un cuvânt (word), ceea ce facilitează implementarea. Dacă dimensiunile parametrilor variabili ar fi diferite, ar trebui folosit operatorul de conversie de tip PTR, pentru a încărca parametrul în mod corect.

Pentru afișarea propriu-zisă se vor folosi macroinstrucțiunile de apel puts (șiruri), puti (întregi cu semn), putu (întregi fără semn) și putc (afișare caracter), descrise în Capitolul 2.

Procedura și toate adresele implicate se consideră de tip NEAR. În Figura 6.10 este descris conținutul stivei la un apel tipic al acestei proceduri.

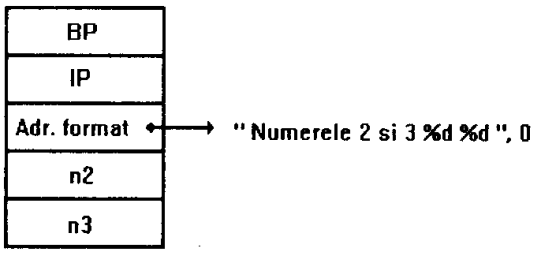


Figura 6.10 Conținutul stivei la intrarea în printf_proc

Pentru accesul la stivă, se va utiliza șablonul:

```
sablon struc
      _bp      dw ?
      _ip      dw ?
```

```
        format    dw ?              ; Adresa sir format
        par       dw ?              ; Primul parametru
    sablon ends
```

Procedura va consta dintr-o buclă de analiză a caracterelor din șirul format, a cărui adresă e gestionată în BX. Astfel, [BX] va reprezenta caracterul curent. Pentru specificatorii de format, se gestionează un contor de parametri variabili, în registrul DI, care va lua valorile 0, 2, 4 etc. (toți parametrii variabili sunt pe 16 biți). Accesul la parametrii variabili se va face prin expresia [BP][DI].PAR, deci printr-o adresare bazată și indexată, cu deplasament. Se tratează corespunzător secvențele escape (care încep cu \) și specificatorii de format (care încep cu %), ambele situații presupunând citirea unui caracter suplimentar din șirul care descrie formatul. Dacă parametrii nu ar avea toți aceeași dimensiune, s-ar utiliza expresii de genul DWORD PTR [BP][DI].PAR, caz în care registrul DI ar fi actualizat cu dimensiunea parametrului (în cazul de față, cu 4).

Implementarea este următoarea:

```
printf_proc proc  near
      push  bp                      ; Salvare BP
      mov   bp,sp
      mov   bx, [bp].format         ; Adresa sir format
      mov   di, 0                   ; Contor parametri
                                    ; variabili
pr_loop:
      mov   al, [bx]                ; Caracter curent
      test  al, al                  ; Test sfarsit de sir
      jnz   aici1                   ; care descrie formatul
      jmp   pr_end
aici1:
      cmp   al, '%'                 ; Test spec. format
      jne   pr_char                 ; Nu este spec.
      inc   bx                      ; Daca da, se citeste
      mov   al, [bx]                ; caracterul urmator
      cmp   al, 's'                 ; Este %s ?
      je    pr_str                  ; Da, salt la afisare
      cmp   al, 'd'                 ; Este %d ?
      je    pr_int                  ; Da, salt la afisare
      cmp   al, 'u'                 ; Este %u ?
      je    pr_u                    ; Da, salt la afisare
      cmp   al, '%'                 ; Este %% ?
      je    aici2 ;
      jmp   pr_err                  ; Nu a fost nici unul
                                    ; din specificatorii
                                    ; asteptati: se afiseaza
                                    ; un mesaj de eroare
aici2:
      putc  al                      ; Afisare caracter
      inc   bx                      ; Avans adresa sir
      jmp   pr_loop                 ; Reluare
pr_char:
      cmp   al, '\'                 ; Este secventa escape ?
      je    pr_esc                  ; Da, salt la tratare
```

```
        putc    al                      ; Daca nu, e vorba de
                                        ; un car. obisnuit, care
        inc     bx                      ; se afiseaza si se reia
        jmp     pr_loop                 ; bucla de parcurgere
pr_esc:
        inc     bx                      ; Tratare secventa escape
        mov     al, [bx]                ; Luam urmatorul caracter
        cmp     al, 'n'                 ; Este \n ?
        jne     esc1                    ; Nu, salt
        putc    cr                      ; Da, se afiseaza
        putc    lf                      ; CR LF si se reia
        inc     bx                      ; bucla de
        jmp     pr_loop                 ; parcurgere
esc1:
        cmp     al, '\'                 ; Este \\ ?
        jne     pr_err                  ; Nu, secventa escape
                                        ; gresita
        putc    al                      ; Da, afiseaza \
        inc     bx                      ; si reia bucla
        jmp     pr_loop                 ; de prelucrare

        ;
        ; Afisarea parametrilor variabili
        ;
pr_str:
        push    bx                      ; Afisare sir. Preluam
        mov     bx, [bp+di].par         ; adresa din stiva
        puts    [bx]                    ; si afisam sirul (cu
        pop     bx                      ; salvare/restaurare BX
pr_next:
        inc     bx                      ; Car. urmator in format
        add     di, 2                   ; Contor par. variabili
        jmp     pr_loop                 ; Reluare
pr_int:
        puti    [bp+di].par             ; intreg cu semn
        jmp     pr_next                 ; Reluare
pr_u:
        putu    [bp+di].par             ; intreg fara semn
        jmp     pr_next                 ; Reluare
pr_err:
                                        ; Mesaj de eroare
        putsi <cr, lf, 'printf: Eroare format', cr, lf>
pr_end:
        pop     bp                      ; Refacere BP
        ret                             ; Revenire
printf_proc endp
```

Ne propunem acum să dezvoltăm o macroinstrucțiune de apel, numită printf, care să se apropie cât mai mult de un limbaj de nivel înalt, deci să putem scrie în textul sursă ceva de genul:

```
printf <"Numerele m n si p sunt: %d %d\n">, <m, n, p>
```

Macroinstrucțiunea va avea, deci, un prim parametru (șirul care descrie formatul) și o listă de parametri variabili (care poate chiar lipsi):

```
printf macro format, param
```

Scrierea unei astfel de macroinstrucțiuni ridică probleme deosebite, dar pe care operatorii și directivele limbajului de asamblare le pot rezolva.

Șirul care descrie formatul se definește în segmentul de date, printr-o tehnică similară celei de la macroinstrucțiunea putsi, descrisă în Capitolul 5.

Problema cea mai dificilă este că parametrii variabili trebuie puși în stivă în ordine inversă decât apar în lista din apelul macroinstrucțiunii. Lista poate fi parcursă cu instrucțiunea IRP, dar numai de la stânga la dreapta.

Pentru a pune parametrii în ordinea dorită, se va parcurge lista lor de două ori. La prima parcurgere, se contorizează numărul de parametri (cu operatorul de atribuire =) și se creează spațiu în stivă, prin secvența:

```
N = 0
irp    x, <param>
       sub  sp, 2
       N = N + 1
endm
```

La a doua parcurgere, se plasează parametrii în stivă. Problema este că indicatorul SP trebuie să fie incrementat cu 2, deci nu se poate utiliza o simplă instrucțiune PUSH. Se execută deci secvența:

```
irp    x, <param>
       add   sp, 2
       push  x
       add   sp, 2
endm
```

care pune parametrii x în stivă, de sus în jos (vezi Figura 6.11).

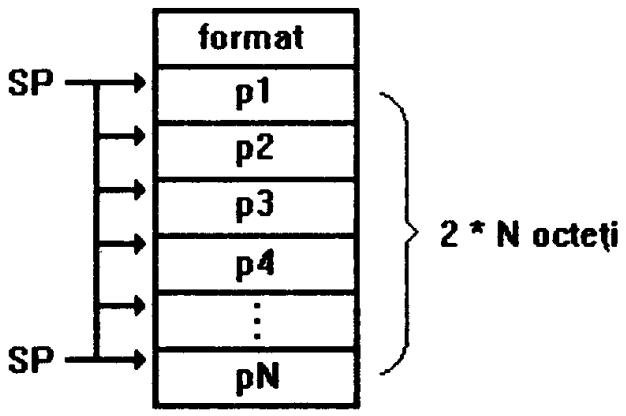


Figura 6.11 Plasarea parametrilor în stivă la macroinstrucțiunea printf

Toată această secvență se execută numai dacă lista param este nevidă. După depunerea parametrilor variabili, se decrementează SP cu 2*N (deci se aduce SP pe primul parametru), se depune în stivă adresa formatului și se apelează procedura printf_proc. Contorizarea parametrilor în constanta simbolică N asigură și o descărcare comodă a stivei, prin adunarea valorii 2*(N+1). Dacă șirul format lipsește, se forțează o eroare de asamblare.

Implementarea completă a macroinstrucțiunii este următoarea:

```
printf macro format, param
local loc_for
ifb   <format>
      %out : Lipsa format....
      .err
      .exitm
endif
.data
loc_for     db    format       ;; Definire sir format
            db    0
.code
N = 0                          ;; Contor par. variabili
ifnb  <param>                  ;; Daca exista par. var.
      irp   x, <param>
            sub   sp, 2        ;; Creare spatiu
            N = N + 1          ;; Contor
      endm

      irp   x, <param>
            add   sp, 2        ;; in jos
            push  x            ;; in sus
            add   sp,2         ;; in jos
      endm
      sub   sp, 2*N            ;; SP pe primul par. var.
endif
lea         ax, loc_for        ;; Adresa format
push        ax                 ;; Depunere in stiva
call        printf_proc        ;; Apel
add         sp, 2*(N+1)        ;; Descarcare stiva
endm
```

Să considerăm un exemplu de apel, în care avem definite datele:

```
.data
      n1    dw    1234
      n2    dw    -3
      n3    dw    4321
      sir1  db    "abcdef",0
      sir2  db    "1234567890",0
      sir3  db    "Acesta e un sir...",0
      s1    dw    sir1
      s2    dw    sir2
      s3    dw    sir3
```

Secvența de test (programul principal) este următoarea:

```
.code
start:
      init_ds_es
      printf <"Numarul 1: %d\n">, <n1>
      printf <"Numerele 2 si 3: %d %d">, <n2,n3>
      printf <"\nSirul 1: %s %% \\ %%">, <s1>
      printf <"\nSirul 2: %s\nSirul 3: %s">, <s2,s3>
      printf <"\nDoar format (fara descriptori) \n">
      printf  <"n2 cu semn = %d, n2 fara semn = %u\n">, <n2,n2>
      exit_dos
end start
```

Se observă necesitatea definirii unor variabile pointer (s2, s2, s3) care conțin adresele șirurilor de caractere sir1, sir2 și sir3. Rularea acestui exemplu produce ieșirea la consolă:

```
Numarul 1: 1234
Numerele 2 si 3: -3 4321
Sirul 1: abcdef % \ %
Sirul 2: 1234567890
Sirul 3: Acesta e un sir...
Doar format (fara descriptori)
n2 cu semn = -3, n2 fara semn = 65533
```

Mai trebuie observat că tehnica de transfer a parametrilor variabili, expusă mai sus, asigură și o bună protecție a controlului programului în situații de eroare, adică atunci când descriptorii de format nu coincid cu parametrii, atât ca număr, cât și ca tip. Să considerăm, de exemplu, apelurile:

```
printf <"Ceva gresit %d %d %d\n">, <n1, n2>
printf <"Ceva si mai gresit %d %d\n">, <n1, s2, s3>
```

În primul caz, formatul precizează că în stivă sunt trei parametri de tip întreg, dar la apel nu se transmit decât doi. Procedura va afișa un al treilea parametru fictiv, dar, deoarece stiva este descărcată de programul apelant, revenirea se face corect și programul nu se distruge. În al doilea caz, primul parametru se afișează corect, al doilea (o adresă de șir) este afișat incorect ca întreg, iar al treilea este pur și simplu ignorat. Și în acest caz, revenirea în programul apelant și continuarea sa nu sunt compromise.

6.9 Tehnici avansate de programare

În subcapitolele anterioare au fost descrise elementele de bază ale programării în ASM (acțiunile principale din programarea structurată, proceduri, definiții de date etc.). Cu aceste elemente de bază, se pot aplica orice tehnici de programare cunoscute de la limbaje de nivel înalt. Vom exemplifica o asemenea tehnică, anume cea a automatelor de stare.

Considerăm problema elementară a numărării cuvintelor dintr-un șir de caractere. Se consideră o serie de caractere speciale, numite delimitatori (în cazul de față: spațiu, punct, virgulă, semn de întrebare, semn de exclamare și cratimă). Evident, se pot specifica orice fel de delimitatori.

Incrementarea numărului de cuvinte nu depinde numai de caracterul curent din șir. De exemplu, un caracter care nu este delimitator va conduce la incrementarea numărului de cuvinte numai în cazul în care caracterul anterior a fost un delimitator. Acest tip de decizie, care se bazează pe o informație anterioară, sugerează utilizarea unei variabile de stare, care să memoreze starea curentă a prelucrării.

În cazul de față, vom avea două stări: în interiorul unui cuvânt și în exteriorul unui cuvânt. Notăm cele două stări cu IN_W și OUT_W. Pentru memorarea lor, este suficientă o variabilă de stare binară.

Algoritmul se modelează ca un automat, care acceptă intrări și generează ieșiri. Funcție de intrări și de starea curentă, se trece într-o stare următoare, emițând sau nu o ieșire. În cazul de față, intrările sunt:

- in_1 = caracterul curent din șir nu este delimitator;
- in_2 = caracterul curent din șir este delimitator.

Ieșirea automatului este:

- out = incrementează numărul de cuvinte

Trebuie acum să codificăm stările, intrările și ieșirile. Pentru simplitate, vom utiliza registrele procesorului:

- stare = IN_W: CX = 1
- stare = OUT_W: CX = 0
- in_1: BX = 1
- in_2: BX = 0
- out: Incrementează AX.

Descriem acum funcționarea automatului, prin diagrama din Figura 6.12.

Dacă suntem în interiorul unui cuvânt (starea IN_W), atunci un delimitator (intrarea in_2) provoacă trecerea în exteriorul unui cuvânt (starea OUT_W), iar un caracter care nu este delimitator (intrarea in_1) menține starea IN_W. Dacă suntem în exteriorul unui cuvânt (starea OUT_W), atunci un delimitator (intrarea in_2) menține starea OUT_W, iar un caracter care nu este delimitator (intrarea in_1) provoacă trecerea în starea IN_W și forțarea semnalului de ieșire out. Inițial, automatul se află în starea OUT_W. Se remarcă faptul că, în codificarea aleasă, starea următoare este dictată de codificarea intrării (mai precis, comutarea stării va însemna o simplă instrucțiune MOV CX, BX).

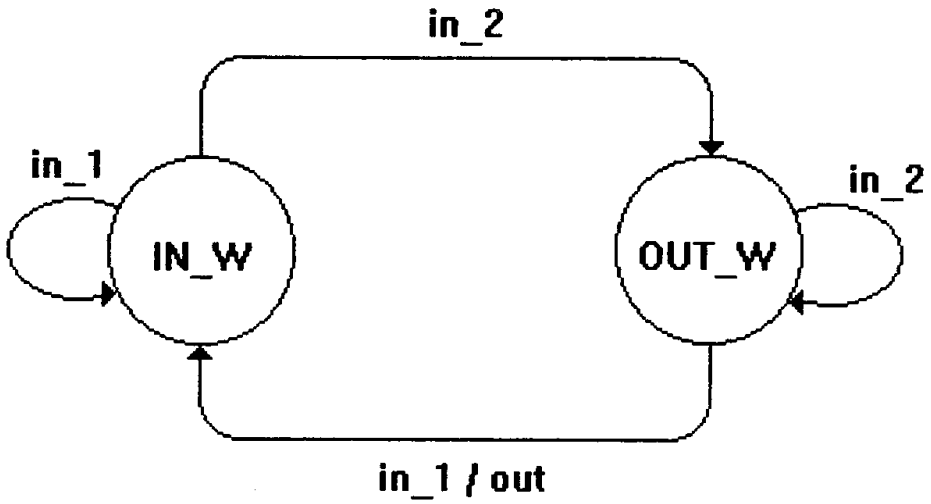


Figura 6.12 Descrierea funcționării unui automat de stare

Evoluția variabilei de stare, a intrărilor in_1 și in_2, precum și a ieșirii out în cazul prelucrării șirului de caractere „Un text oarecare." sunt ilustrate în Figura 6.13.

Implementarea algoritmului urmărește îndeaproape funcționarea automatului și constă în principal din:

- evaluarea intrărilor;
- comutarea stării;
- forțarea ieșirii.

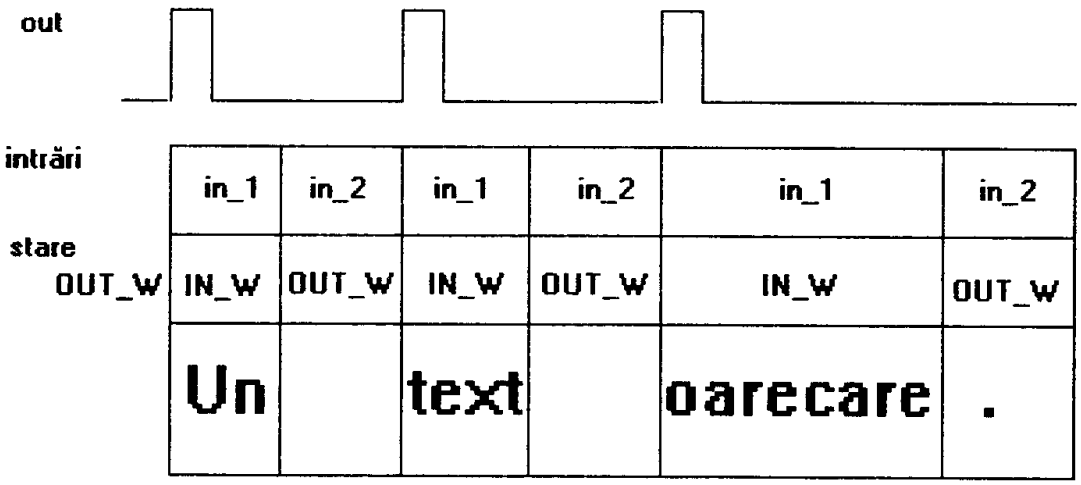


Figura 6.13 Evoluția automatului de stare în cazul șirului „Un text oarecare."

Presupunem că adresa șirului este transmisă procedurii în DS:SI și că rezultatul este întors în AX. Implementarea este următoarea:

```
count proc near
      push  cx                        ; Salvari
      push  bx                        ; registre
      push  si
      mov   cx, 0                     ; Stare initiala
                                      ;  = OUT_W
      mov   ax, cx                    ; Numar de cuvinte = 0
reluare:
      cmp   byte ptr [si], 0          ; Test sfarsit de sir
      je    gata
      cmp   byte ptr [si], ' '        ; Evaluare intrari:
                                      ; Blank
      je    et_1
      cmp   byte ptr [si], tab        ; Tab
      je    et_1
      cmp   byte ptr [si], '.'        ; Punct
      je    et_1
      cmp   byte ptr [si], ','        ; Virgula
      je    et_1
      cmp   byte ptr [si], '?'        ; Semn de intrebare
      je    et_1
      cmp   byte ptr [si], '!'        ; Semn de exclamare
      je    et_1
      cmp   byte ptr [si], '-'        ; Cratima
      je    et_1

      mov   bx, 1                     ; Nu e delimitator
      jmp   et_2
et_1:
      mov   bx, 0                     ; Delimitator
;
;     in acest moment, intrarile sunt pozitionate
;     BX = 1 (in_1) sau BX = 0 (in_2)
et_2:
      cmp   cx, 0                     ; Stare = OUT_W ?
      je    et_3
      mov   cx, bx                    ; Stare = intrare
      jmp   et_5
et_3:
      cmp   bx, 1                     ; In starea OUT_W si
      jne   et_4                      ; cu intrarea in_1,
      inc   ax                        ; se genereaza iesirea
                                      ; out (incrementeaza
                                      ; numarul
                                      ; de cuvinte)
et_4:
      mov   cx, bx                    ; Stare = intrare
et_5:
      inc   si                        ; Trecere la
      jmp   reluare                   ; caracterul urmator
gata:
      pop   si                        ; Refaceri
```

```
        pop     bx                      ; registre
        pop     cx
        ret                             ; Rezultat in AX
count   endp
```

Un program de test al procedurii count este listat mai jos. Se citesc de la consolă șiruri de caractere (până la introducerea șirului vid), se apelează procedura count și se afișează numărul de cuvinte din șirul introdus.

```
.model      small
include     io.h
.data
        sir     db      80 dup (0)      ; Spatiu de lucru
.stack 1024
.code

start:
        init_ds_es
iar:
        putsi <'Introduceti un sir: '>
        gets    sir                     ; Citire sir
        mov     al, sir                 ; Test sir
        test    al, al                  ; vid
        jz      exit
        putsi <'Sirul are '>
        lea     si, sir                 ; Apel procedura
        call    count                   ; Nr. cuvinte in AX
        puti    ax                      ; Afisare
        putsi <' cuvinte', cr, lf>
        jmp     iar                     ; Reluare
exit:
        exit_dos
end     start
```

6.10 Programarea coprocesoarelor matematice

În Capitolul 2, a fost prezentat setul de instrucțiuni al coprocesoarelor matematice 80x87. Coprocesorul recunoaște variabile întregi și toate cele trei tipuri de numere reale (precizie simplă, dublă și extinsă).

În acest subcapitol, ne propunem să dezvoltăm un set de proceduri și macroinstrucțiuni care să rezolve problema conversiei din formatul real (în simplă precizie), în format ASCII (șir de caractere), ceea ce va permite apoi și introducerea și afișarea de numere reale la consolă.

6.10.1 Conversia ASCII-real

Dezvoltăm o procedură de tip far, cu numele ATOF_PROC, care primește în stivă o adresă near a unui șir de caractere și o adresă near a unui număr real. Șablonul de acces la stivă este:

```
sablon_atof struc
                  dw    ?      ; Loc pentru BP
                  dd    ?      ; Loc pentru adresa de revenire
      adr_num     dw    ?
      adr_sir     dw    ?
sablon_atof ends
```

Procedura de conversie se bazează pe următorul algoritm:

```
      valoare = 0;
      divizor = 1.0
      punct = FALSE;
      minus = FALSE;
      preia caracter din sirul sursa;
      if (caracter == '-' ) {
            minus = TRUE;
            preia urmatorul caracter din sirul sursa;
      }
      while (caracterul curent este cifra zecimala sau '.') {
            if (caracter == '.')
                  punct = TRUE;
            else {
                  cifra = caracter - '0';
                  cifra_float  = (float) cifra;
                  valoare = 10.0 * valoare + digit_float;
                  if (punct)
                        divizor = divizor * 10.0;
            }
            preia urmatorul caracter din sir;
      }
      if (punct)
            valoare = valoare/divizor;
      if (minus)
            valoare = - valoare;
      return valoare;
```

Ideea generală este calculul numărului ca și când nu ar avea punctul zecimal și împărțirea apoi la o putere a lui 10, calculată după numărul de zecimale. De exemplu, șirul 123.456 va conduce la calculul valorii 123456, care se împarte apoi la 1000.

Implementarea este următoarea:

```
.data
      _zece       dd    10.0
      _punct      db    ?
      _minus      db    ?
      _cifra      dw    ?
.code
atof_proc proc far
      push  bp
      mov   bp, sp
      push  bx                        ; Salvari
      push  si                        ; registre
```

```
        fld1                            ; ST <- 1.0
        fldz                            ; ST <- 0.0, ST(1) <- 1.0
;
;       Asignam valoare in ST si divizor in ST(1)
;
        mov     _punct, 0
        mov     _minus, 0
        xor     bh, bh                  ; Necesar la depunere intreg
        mov     si, [bp].adr_sir        ; Adresa sir
        cmp     byte ptr [si], '-'      ; Test '-'
        jne     atof_1
        mov     _minus, 1
        inc     si
atof_1:
        mov     bl, [si]                ; Preia caracter
        cmp     bl, '.'                 ; Test '.'
        jne     atof_2
        mov     _punct, 1
        jmp     atof_3                  ; Salt la reluare
atof_2:
        cmp     bl, '0'                 ; Test
        jb      atof_4                  ; cifra
        cmp     bl, '9'                 ; zecimala
        ja      atof_4
        sub     bl, '0'                 ; Conversie la intreg
        mov     _cifra, bx              ; Memoram ca intreg
        fmul    _zece                   ; valoare = 10*valoare
        fiadd   _cifra                  ; valoare = valoare +
                                        ; cifra
        cmp     _punct, 1               ; Test punct
        jne     atof_3
        fxch                            ; Schimbam ST cu ST(1)
        fmul    _zece                   ; pentru ca vrem
                                        ; sa inmultim
        fxch                            ; divizorul cu zece,
                                        ; apoi
                                        ; schimbam la loc
atof_3:
        inc     si                      ; Reluare
        jmp     atof_1                  ; bucla
atof_4:
        fdivr                           ; impartire ST la ST(1)
                                        ; cu descarcarea stivei
                                        ; Rezultat in ST
        cmp     _minus, 1               ; Test semn
        jne     atof_5
        f_comp  _zero                   ; Daca e 0.0 nu
        je      atof_5                  ; schimba semnul
        fchs                            ; Schimbare semn
atof_5:
        mov     bx, [bp].adr_num        ; Adresa numar real
        fstp    dword ptr [bx]          ; Depunere ST
```

```
                                        ; cu descarcarea
        fwait                           ; stivei, apoi FWAIT

        pop     si                      ; Refaceri
        pop     bx                      ; registre
        pop     bp
        retf                            ; Revenire
atof_proc endp
```

Se observă că stiva coprocesorului este lăsată în starea în care era la intrarea în procedură, lucru care trebuie avut în vedere la toate programele care lucrează cu 80x87. Într-o implementare mai pretențioasă, s-ar salva toate cele 8 registre ale coprocesorului ca variabile de tip TBytes, într-o zonă proprie de memorie și s-ar reface la revenirea în programul apelant.

De asemenea, bucla de citire caractere ar trebui să testeze dacă s-a întâlnit deja punctul zecimal. Pentru simplitate, s-a presupus aprioric că șirul sursă conține cel mult un punct zecimal.

Scriem acum o macroinstrucțiune de apel, cu numele atof, care are ca parametri șirul de caractere sursă și variabila reală în care se va depune rezultatul. Ambii parametri sunt folosiți în instrucțiuni LEA, care preiau adresele near și le transmit către procedura atof_proc.

```
atof  macro a_sir, a_num
      push  si
      lea   si, a_sir
      push  si
      lea   si, a_num
      push  si
      call  far ptr atof_proc
      add   sp, 4
      pop   si
endm
```

Dispunând de macroinstrucțiunea atof, putem scrie o macroinstrucțiune cu numele getf, care să citească un număr real de la consolă. Operația se compune dintr-o citire de șir de caractere și o conversie ASCII-real. Parametrul macroinstrucțiunii este variabila dword în care se depune numărul real citit.

```
getf  macro       a_num
local sir
      .data
            sir   db 80 dup (0)
      .code
            gets  sir
            atof  sir, a_num
endm
```

6.10.2 Compararea numerelor reale

O problemă importantă care se pune este compararea a două numere reale. Procesorul dispune de instrucțiuni de comparare, în urma cărora se poziționează o serie de flaguri interne din cuvântul de stare (Status Word). Acest cuvânt poate fi depus în memorie cu instrucțiuni FSTSW și citit apoi prin instrucțiuni 80x86. Preferăm totuși o altă abordare, mai simplă, anume efectuarea diferenței celor doi operanzi și poziționarea corespunzătoare a bistabililor ZF și CF din 80x86. Astfel, se vor putea utiliza instrucțiuni de salt condiționat specifice 80x86 (JE, JB, JGE etc.). Testarea semnului diferenței se face prin examinarea fizică a bitului de semn din reprezentarea internă a numerelor în simplă precizie. Testarea cazului de egalitate se face testând dacă toți cei patru octeți ai diferenței sunt nuli, ignorând bitul de semn.

Dezvoltăm macroinstrucțiunea F_COMP, cu un operand, care compară numărul real din vârful stivei coprocesorului cu operandul specificat.

```
f_comp macro      val
      local       temp, etgt, etlt, eteq, gata
.data
      temp  dd    ?                    ; Spatiu de lucru
.code
      push  ax                         ; Salvare AX
      fld   st                         ; Se face o copie a varfului
                                       ; stivei, pentru a nu
                                       ; altera ST initial
      fsub  dword ptr val              ; ST <- ST - val
      fstp  dword ptr temp             ; Depunem diferenta
      mov   al, byte ptr temp+3        ; Luam ultimul octet
                                       ; din reprezentare
      and   al, 10000000B              ; Filtru bit de semn
      jnz   etlt                       ; Diferenta negativa ?
      mov   ax, word ptr temp          ; Nu, testam daca nu
                                       ; este zero
      or    ax, word ptr temp+2        ; Zero real are toti cei
                                       ; 4 octeti nuli
      jz    eteq                       ; Este zero ?
etgt: mov   ax, 2                      ; Nu, inseamna ca e mai mare
      cmp   ax, 1                      ; Facem o comparatie pentru a
      jmp   gata                       ; pozitiona indicatorii
etlt: mov   ax, word ptr temp
      or    ax, word ptr temp+2
      and   ax, 7FFFH                  ; -0 si 0 sunt identice
      jz    eteq
      mov   ax, 1                      ; Cazul mai mic
      cmp   ax, 2
      jmp   gata
eteq: mov   ax, 1                      ; Cazul egal
      cmp   ax, 1
gata: pop   ax                         ; Refacere AX si gata
endm
```

6.10.3 Conversia real-ASCII

Dezvoltăm acum procedura ftoa_proc, care primește în stivă adresa near a unui număr real și adresa near a unui șir de caractere. Procedura convertește numărul real într-un șir de caractere care conține reprezentarea valorii reale în format științific:

x.xxxxxxE±yy

Accesul la parametrii din stivă se face prin structura șablon:

```
sablon_ftoa struc
                  dw    ?
                  dd    ?
      val         dd    ?
      siradr      dw    ?
sablon_ftoa end
```

Implementarea folosește o serie de constante și variabile de lucru, definite mai jos. Algoritmul de conversie este următorul:

```
;
;     val = Valoarea reala care se converteste
;     sir = Adresa sirului destinatie
;
if (val < 0.0) {
      *s++ = '-';
      val = -val;
}
_exp = 0;
if (val != 0.0) {
      if (val >= 0.0) {
            do {
                  val = val / 10.0;
                  _exp = _exp + 1;
            } while (value >= 10.0)
      }
      else {
            while (val < 1.0) {
                  val = val * 10.0;
                  _exp = _exp - 1;
            }
      }
      val = val + 0.0000005;
      if (value > 10.0) {
            value = value / 10.0;
            _exp = _exp + 1;
      }
}
_cifra = (int) val; /* Obligatoriu cu trunchiere, nu cu
                       rotunjire */
*s ++ = _cifra + '0';
*s++ = '.';
for (i = 1; i <= 6; i++) {
      val = 10.0 * (val - (float) _cifra);
```

```
        _cifra = (int) val; /* Obligatoriu cu trunchiere, nu
                               cu rotunjire */
        *s++ = _cifra + '0';
    }
    *s++ = 'E';
    if (_exp < 0) {
        *s++ = '+';
        _exp = - _exp;
    }
    else
        *s++ = '+';
                        /* scrie _exp ca două cifre în sir */
    *s = 0;             /* Terminator de sir */
```

Se aduce numărul în intervalul [1.0, 10.0), prin împărțiri sau înmulțiri succesive cu 10. Înmulțirile sau împărțirile cu 10 sunt contorizate în variabila întreagă _exp (exponent zecimal). În acest moment, partea întreagă a numărului are o singură cifră, care se poate extrage și depune în șir. Se determină partea fracționară, prin scăderea părții întregi, se înmulțește cu 10 și se reia generarea cifrelor. Acest proces se repetă de 5 ori, pentru cifrele de după virgulă.

În prealabil, se realizează o rotunjire a numărului, prin adunarea valorii reale 0.000005.

Algoritmul depinde esențial de modul în care se face conversia real-întreg. Este obligatoriu ca această conversie să se realizeze prin trunchiere, nu prin rotunjire. De exemplu, valoarea 2.999 trebuie să genereze prima cifră 2, și nu 3.

Coprocesorul poate opera în mai multe moduri de rotunjire/trunchiere. Aceste moduri sunt controlate de biții 10 și 11 (câmpul RC sau Round Control), din cuvântul de control (Control Word). Pentru situația de față, este adecvat modul RC = 1. Procedura trebuie deci să poziționeze câmpul RC din cuvântul de control, printr-o salvare/încărcare în memorie. La revenirea în programul apelant, se va reface câmpul RC original.

Procedura necesită o serie de constante reale și variabile de manevră, definite mai jos. Numele variabilelor coincid cu cele de la descrierea algoritmului. Evident, vom aloca variabila reală în vârful stivei 80x87. Exponentul este gestionat ca număr întreg.

```
.data
        _zece   dd    10.0
        _unu    dd    1.0
        _zero   dd    0.0
        _round  dd    0.0000005
        _exp    dw    ?
        _cifra  dw    ?
        _temp   dd    ?
        _cw     dw    ?
```

Implementarea propriu-zisă este următoarea:

```
.code
ftoa_proc proc far
      push  bp
      mov   bp, sp

      push  ax                              ; Salvare
      push  bx                              ; registre
      push  si
      push  cx

      fstcw  _cw                            ; Salvare Control Word
      mov    ax, _cw
      and    ax, NOT 0000110000000000B      ; Filtru bitii 10 si 11
      or     ax, 0000010000000000B          ; Fortare RC = 1
      mov    _cw, ax                        ; inapoi in memorie
      fldcw  _cw                            ; inapoi in 80x87

      mov    _exp, 0                        ; Exponent
      mov    si, [bp].siradr                ; Adresa sir
                                            ; de caractere
      fld    dword ptr [bp].val             ; Valoare reala în ST
      f_comp _zero
      jg     ftoa_poz                       ; Este > 0 ?
      mov    byte ptr [si], '-'             ; Nu, depunem '-'
      inc    si                             ; in sir si schimbam
      fchs                                  ; semnul numarului real
ftoa_poz:
      f_comp _zero
      jne    ftoa_aici                      ; Este diferit de 0.0 ?
      jmp    ftoa_1
ftoa_aici:                                  ; Da, il convertim
      f_comp _zece
      jnge   ftoa_2                         ; Este < 10 ?
ftoa_3:
      fdiv   _zece                          ; Nu, il impartim la 10
      add    _exp, 1                        ; si tinem minte
                                            ; la exponent
      f_comp _zece                          ; pana cand ajunge
      jnl    ftoa_3                         ; mai mic strict
                                            ; decat 10
ftoa_2:
      f_comp _unu                           ; Este mai mare
                                            ; sau egal ca 1 ?
      jge    ftoa_1                         ; Da, salt
      fmul   _zece                          ; Nu, il inmultim cu
                                            ; 10 si tinem minte
      sub    _exp, 1                        ; la exponent,
                                            ; pana ajunge
      jmp    ftoa_2                         ; mai mare sau
                                            ; egal ca 1
ftoa_1:
      fadd   _round                         ; Rotunjim la
                                            ; 7 zecimale
```

```
        f_comp _zece                    ; Poate a depasit
                                        ; acum 1o ?
        jna    ftoa_4
        fdiv   _zece                    ; Daca da, corectam
        add    _exp, 1
ftoa_4:
        fist   _cifra                   ; Generam partea
                                        ; intreaga
        mov    bl, byte ptr _cifra      ; Luam cifra
                                        ; propriu-zisa,
        add    bl, '0'                  ; o convertim la
                                        ; caracter ASCII
        mov    [si], bl                 ; si o depunem in sir
        inc    si
        mov    byte ptr [si], '.'       ; Apoi punctul zecimal
        inc    si

        mov    cx, 6                    ; Bucla de 6 cifre
                                        ; dupa punct
ftoa_5:
        fisub  _cifra                   ; ST = partea
                                        ; fractionara
        fmul   _zece                        ; ST = ST * 10
        fist   _cifra                   ; Partea intreaga
        mov    bl, byte ptr _cifra      ; Determinare cifra
        add    bl, '0'
        mov    [si], bl
        inc    si
        loop   ftoa_5

        fstp   _temp                    ; Descarcam stiva 8087
        mov    byte ptr [si], 'E'       ; Notatie stiintifica
        inc    si
        cmp    _exp, 0                  ; Test semn exponent
        jge    ftoa_6
        mov    byte ptr [si], '-'       ; Negativ
        neg    _exp                     ; Schimbam semnul
        jmp    ftoa_7
ftoa_6:
        mov    byte ptr [si], '+'       ; Pozitiv
ftoa_7:
        inc    si
        mov    ax, _exp                 ; Exponent
        aam                             ; AH = _exp / 10;
                                        ; AL = _exp MOD 10
        or     ax, 3030H                ; Ambele cifre
                                        ; convertite la ASCII
        mov    [si], ah                 ; Depunere in sir
        inc    si
        mov    [si], al
        inc    si
        mov    byte ptr [si], 0         ; in fine, terminatorul
                                        ; de sir

        pop    cx                       ; Refacere
```

```
        pop     si                              ; registre
        pop     bx
        pop     ax

        pop     bp
        retf                                    ; Revenire
ftoa_proc       endp
```

Se observă utilitatea foarte mare a macroinstrucțiunii f_comp, care face ca implementarea algoritmului să fie o simplă transpunere a descrierii de tip pseudo-cod.

Dezvoltăm acum o macroinstrucțiune de apel cu numele ftoa și cu parametri adecvați (valoarea reală și șirul destinație):

```
ftoa    macro           val, sir
        push    ax
        lea     ax, sir
        push    ax
        mov     ax, word ptr val + 2
        push    ax
        mov     ax, word ptr val
        push    ax
        call    far ptr ftoa_proc
        add     sp, 6
        pop     ax
endm
```

Dispunând de macroinstrucțiunea ftoa, putem dezvolta o macroinstrucțiune putf, care să afișeze la consolă un număr real, prin conversie real-ASCII și afișare șir:

```
putf    macro val
        local           sir
        .data
                sir     db 30 dup (0)
        .code
                ftoa    val, sir
                puts    sir
endm
```

6.10.4 Un program de test

Dacă presupunem macroinstrucțiunile de mai sus definite într-un fișier float.h, putem scrie un mic program de test, care citește un număr real de la consolă și afișează valorile incrementate de 10 ori.

```
        .model          large
        .8087
        include         io.h
        include         float.h
        .stack          1024
        .data
                numar dd    ?
                unu   dd    1.0
```

```
.code
start:
        init_ds_es
        finit
        putsi <'Introduceti un numar real: '>
        getf  numar
        putsi <'Valorile incrementate sunt:', cr, lf>
        mov   cx, 10
iar:
        putf  numar
        putsi <cr, lf>
        fld   numar
        fadd  unu
        fstp  numar
        loop  iar
        exit_dos
end     start
```

Se observă instrucțiunea **FINIT**, care inițializează coprocesorul matematic. Directiva **.8087** instruiește asamblorul să accepte mnemonice specifice coprocesorului 8087. Funcție de tipul coprocesorului și al procesorului de bază, se pot utiliza directivele **.287**, **.387** sau **.486**.

Procedurile de mai sus operează cu numere în simplă precizie. Trecerea la precizie dublă și extinsă se face însă foarte ușor. Trebuie doar ca variabilele externe și constantele reale utilizate să fie declarate de tip **qword** (8 octeți), respectiv **tbytes** (10 octeți), deoarece formatul intern al procesorului este cel de precizie maximă.

Deși procedurile care implică operații cu numere reale par complicate, implementările din acest subcapitol arată că o abordare sistematică și disciplinată permite dezvoltarea unui cod sursă clar, ușor de urmărit și de întreținut.

Capitolul 7

Interfața ASM - limbaje de nivel înalt

O parte importantă a programării în limbaj de asamblare o constituie legarea modulelor de program ASM cu module de program dezvoltate în limbaj de nivel înalt. În acest mod, se poate crește eficiența aplicațiilor, prin dezvoltarea modulelor critice (care se utilizează foarte des) în ASM.

În acest capitol, va fi exemplificată dezvoltarea de aplicații mixte în limbaj de asamblare și în limbajul C. Alegerea limbajului C ca limbaj reprezentativ de nivel înalt este justificată de următoarele proprietăți:

- răspândirea foarte mare a limbajului C, ca limbaj destinat atât programelor de sistem, cât și celor aplicative;
- existența unor medii de dezvoltare evoluate, total compatibile cu asambloarele 8086 (un exemplu semnificativ fiind familia de produse Borland);
- caracterul modular al limbajului C, prin care se permite compilarea separată a modulelor sursă;
- compatibilitatea totală a modulelor obiect (un modul obiect rezultat în urma compilării unui text sursă C are exact același format ca un modul obiect rezultat al asamblării unui text ASM).

Se va utiliza deci mediul de dezvoltare Borland (compilatorul bcc, editorul de legături tlink, asamblorul tasm, respectiv mediul integrat bc).

7.1 Transferul parametrilor. Simboluri externe

În mod concret, interfața dintre module C și module ASM va consta din apeluri de funcții ASM din C și invers, respectiv din accesul din C la date definite în ASM și invers.

Compilatorul Borland C realizează transferul parametrilor prin stivă, în ordinea de la dreapta la stânga a listei de parametri. Descărcarea stivei este făcută de către modulul apelant. Întoarcerea de tipuri simple de date din funcții se realizează prin registrul acumulator, eventual extins (deci prin AL, AX sau DX:AX, corespunzător unui tip pe 1, 2 sau 4 octeți). Tipurile reale (float, double, long double) sunt întoarse printr-o zonă specială a bibliotecii de virgulă mobilă sau în vârful stivei coprocesorului matematic. Pentru a nu complica interfața, vom considera că funcțiile C folosite în ASM nu întorc valori reale. Dacă este

absolut necesar, se poate adopta soluția ca funcțiile C să întoarcă pointeri la variabile reale statice.

Un alt caz complicat este transferul structurilor și al uniunilor, care este definit în C prin copierea bit cu bit a tuturor membrilor, atât la transfer spre funcție, cât și la întoarcere din funcție. Este și aici de preferat soluția mult mai eficientă de a transfera sau a întoarce un pointer către structura sau uniunea respectivă.

Numele simbolurilor externe în C (funcții și variabile externe) sunt generate implicit cu caracterul _ (subliniere) ca prim caracter. De exemplu, simbolul var este generat ca _var și vizibil ca atare într-un modul ASM. Pentru variabile, numele este sinonim cu adresa unde este memorată variabila.

Deoarece, în C, vizibilitatea externă este permisă numai variabilelor definite la nivel exterior (în afara tuturor funcțiilor), acestea sunt implicit alocate static, deci au adrese fixe de memorie. Dacă, de exemplu, var este un întreg definit în C, atunci el poate fi citit în ASM printr-o instrucțiune de tipul mov ax, _var.

Trebuie cunoscute dimensiunile fizice ale tipurilor de bază din C. Acestea sunt: char (1 octet), int (2 octeți), short (2 octeți), long (4 octeți), float (4 octeți), double (8 octeți), long double (10 octeți), pointeri (2 sau 4 octeți, funcție de modelul de memorie folosit).

Trebuie ținut seama și de faptul că operațiile cu stiva sunt operații pe 16 biți, deci când se transmite un char la o funcție, se pune pe stivă un cuvânt de 16 biți, cu partea mai semnificativă 0. Este recomandabil ca, asemenea funcțiilor din biblioteca C standard, variabilele de tip char să fie transmise și întoarse la și de la funcții ca variabile de tip unsigned int, ceea ce asigură protecție la o eventuală extensie de semn la 16 biți. De exemplu, variabila char '\x81' devine, prin extensie de semn, întregul 0xff81, dar variabila unsigned char '\x81' devine întregul 0x0081. Extensia de semn poate apărea dacă tipul char este implicit signed (ceea ce se întîmplă la Turbo C). Există o opțiune de compilare care face ca tipul char să fie implicit unsigned, dar este bine să se scrie programe care să nu depindă de această opțiune.

Un alt fapt care trebuie avut în vedere este că C-ul este un limbaj de tip case-sensitive, deci contează dacă identificatorii sunt scriși cu litere mici sau mari. Trebuie forțată la TASM opțiunea /ml sau /mx care generează simbolurile ținând cont de acest fapt. Să considerăm următoarea secvență C:

```
include <stdio.h>
int n = 5;
char sir[] = "Un sir";
void main(void)
{
      printf("%s %d\n", sir, n);
}
```

Presupunând modelul de memorie small (adică toate adresele sunt pe 2 octeți), imaginea stivei la intrarea în funcția printf este cea din Figura 7.1.

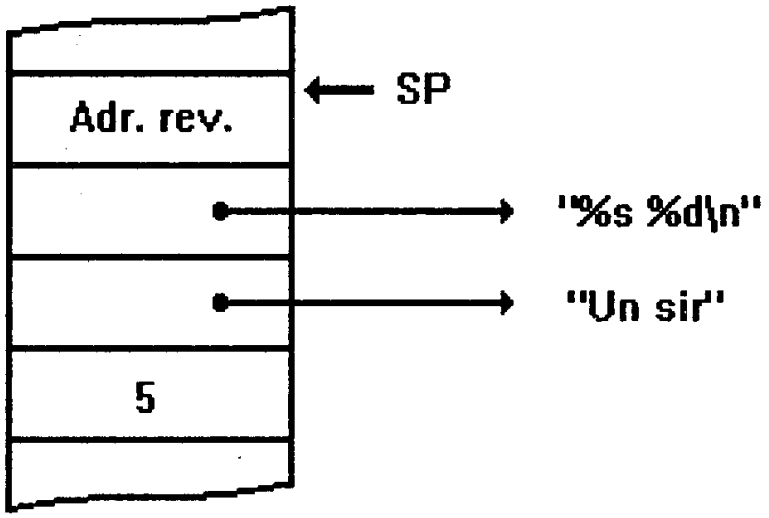


Figura 7.1 Imaginea stivei la un apel al funcției printf

Apelul lui printf din ASM se poate face cu secvența ASM de mai jos (echivalentă cu secvența C):

```
.model small
extrn printf: near
.data
      n            dw    5
      sir          db    'Un sir', 0
      _format      db    '%s %d\n', 0
.code
_main       proc near
      push  _n
      lea   ax, sir
      push  ax
      lea   ax, format
      push  ax
      call  near ptr printf
      add   sp, 6
      retn
_main endp
```

7.2 Dezvoltarea în mediu integrat sau prin linii de comandă

Sunt posibile două moduri de dezvoltare a aplicației: în mediul integrat C sau prin linia de comandă.

În mediul integrat (lansat prin comanda bc) trebuie definit un proiect care să specifice explicit modulele în limbaj de asamblare. Definirea proiectului permite specificarea programului translator (Compiler sau Assembler).

Astfel, se poate lucra în regim de editare, compilare, depanare etc., cu ambele tipuri de module (ASM și C), mediul integrat recunoscând modulele scrise în ASM. Acest context este foarte util pentru depanare, deoarece toate facilitățile specifice mediilor integrate de dezvoltare pentru limbaje de nivel înalt (rulare pas cu pas, vizualizarea permanentă a unor variabile etc.), se aplică și modulelor scrise în ASM.

O altă variantă de depanare o constituie Turbo Debugger-ul, care poate fi apelat direct sau din mediul integrat, acesta recunoscând, de asemenea, atât module ASM, cât și module C.

Opțiunile necesare la compilarea programelor C (cea mai importantă fiind modelul de memorie), se stabilesc prin comanda **Options** a mediului integrat.

A doua posibilitate este compilarea, respectiv asamblarea separată a modulelor sursă, prin lansări ale compilatorului bcc și ale asamblorului tasm de la consolă și apoi legarea explicită a modulelor obiect, prin lansarea editorului de legături tlink. În acest caz, opțiunea de model de memorie pentru modulele C se stabilește în linia de apel a compilatorului (opțiunea -mx, unde x este un caracter ce identifică modelul). De asemenea, trebuie precizată opțiunea -c (compile only), astfel încât compilatorul C să producă numai modul obiect. Tot în linia de comandă se precizează căile pentru fișierele header și pentru biblioteci (opțiunile -I și -L).

Când se lucrează în regim de linie de comandă, trebuie ținut seama de structura unui program C. Pe lângă modulele definite de utilizator, trebuie adăugate modulul de inițializare c0x (se pune pe prima poziție în lista de module obiect la tlink) și bibliotecile C (aflate în subdirectorul \lib), care se specifică după numele fișierului executabil. Atât modulul de inițializare, cât și bibliotecile sunt specifice modelului de memorie folosit, fiind identificate prin ultima literă a numelui fișierului respectiv.

Bibliotecile care trebuie specificate depind de funcțiile utilizate în modulele C. Practic, se va lega totdeauna bibioteca cx.lib și, eventual, mathx.lib, emu.lib (pentru emularea operațiilor în virgulă mobilă), fp87.lib (dacă sistemul are coprocesor), graphics.lib (dacă s-au folosit funcții grafice). În specificarea

numelor fișierelor care conțin modulul de inițializare și bibliotecile respective, caracterul x depinde de modelul de memorie folosit.

Este utilă o vizualizare a conținutului subdirectorului \lib din implementarea Borland C, pentru a identifica bibliotecile C. Este, de asemenea, utilă vizualizarea opțiunilor posibile de compilare, asamblare și link-editare, ca și sintaxa acestor comenzi (se poate obține prin bcc, tasm sau tlink urmate imediat de <Enter>).

În cazul în care modulul de program principal este scris în limbaj de asamblare, trebuie ținut seama de faptul că main este o funcție ca oricare alta. De fapt, modulul de program principal este modulul precompilat c0, care, după ce face o serie de inițializări, apelează procedura main. Din această cauză, dacă funcția main este scrisă în ASM, ea trebuie declarată ca o procedură și specificată ca simbol public:

```
public _main
 _main proc
       ...
       ret
 _main endp
end
```

Procedura _main trebuie încheiată prin instrucțiunea ret, ca orice procedură, iar modulul de program respectiv nu trebuie să conțină etichetă de start. Încheierea execuției se va face la return-ul din _main, controlul revenind modulului c0, care a apelat _main și care este răspunzător de ieșirea în sistemul de operare. Nu este necesară o ieșire în DOS din procedura _main. De asemenea, nu este necesară nici inițializarea registrelor DS și ES, deoarece este făcută în modulul c0. Modulul c0 definește și o stivă minimă (dacă modelul nu este tiny), care se poate mări, dacă este necesar, prin directiva ASM .stack.

În cazul în care funcția cmain este scrisă în C, modulele ASM vor conține, practic proceduri și definiții de date, care trebuie declarate publice (cu ajutorul directivei ASM public). De asemenea, dacă modulele ASM apelează funcții C, acestea trebuie declarate în ASM ca simboluri externe (cu ajutorul directivei ASM extern).

7.3 Modele de memorie

Modelele de memorie corespund organizării segmentelor de date, cod și stivă, specifice adresării de la procesoarele INTEL. Atât asamblorul TASM, cât și compilatorul Borland C recunosc aceleași modele de memorie, generând segmente cu același nume. La TASM, acest lucru corespunde folosirii directivelor de definire simplificată a segmentelor. Această compatibilitate totală simplifică foarte mult efortul de dezvoltare a aplicațiilor mixte.

În ASM, modelul se specifică prin directiva .model, iar în C prin opțiunea Options/Compiler/Code generation.../Model în mediul integrat sau prin opțiunea -mx în linia de comandă, la lansarea compilatorului.

Modelele disponibile sunt tiny, small, compact, medium, large și huge. Funcție de aceste modele, se vor genera apeluri de funcții implicit de tip far sau near și pointeri la funcții sau la date implicit de tip far sau near.

Implementările Borland C permit definirea explicită a pointerilor și a funcțiilor de tip far sau near, dar acest lucru nu este recomandabil. Este indicat să fie lăsat compilatorul să genereze implicit pointeri și apeluri de funcții, conform modelului de memorie definit. Dacă modulele C includ fișierele header predefinite, în care se găsesc prototipurile pentru funcțiile de bibliotecă folosite, apelurile de funcții și legarea cu modulele de bibliotecă se vor realiza în mod corect.

În modulele ASM, se va folosi același model de memorie, dar trebuie ținut cont de tipurile pointerilor și ale funcțiilor definite sau folosite în modulele C, deoarece, în ASM, accesul la date, funcții, adrese de segment, offset-uri etc. se face explicit. Se recomandă ca, în ASM, să se folosească explicit apeluri de tip near/far, definiții de proceduri explicite near/far și return explicit de tip near/far, deși asamblorul generează corect apeluri și definiții de proceduri implicite, pe baza modelului ales. Totuși, pentru a scoate în evidență tipurile respective, este bine să se folosească variante explicite în ASM.

Nu se recomandă în nici un caz ca, în aceeași aplicație, module diferite de program să aibă modele diferite de memorie.

Portabilitatea modulelor C la schimbarea modelului de memorie nu necesită, practic, nici un efort de întreținere. Portabilitatea modulelor ASM față de modelele de memorie este însă o problemă dificilă. Apelurile de proceduri și instrucțiunile de revenire pot fi gestionate corect de către asamblor. Restul operațiilor trebuie gestionate explicit, eventual prin directive de asamblare condiționată.

Să presupunem că transmitem unei proceduri un parametru de tip adresă de variabilă (de date) și dorim să definim o structură de acces la stivă, care să nu depindă de modelul de memorie utilizat. Soluția constă în folosirea directivelor de asamblare condiționată și a simbolului predefinit @MODEL (vezi 4.2).

```
_TINY_      equ 1
_SMALL_     equ 2
_MEDIUM_    equ 3
_COMPACT_   equ 4
_LARGE_     equ 5
_HUGE_      equ 6
DATE_MARI   equ (@MODEL EQ COMPACT) OR (@MODEL EQ LARGE)
COD_MARE    equ (@MODEL EQ MEDIUM) OR (@MODEL EQ LARGE)
DATE_MICI   equ NOT (DATE_MARI)
COD_MIC     equ NOT (COD_MARE)
```

```
      sablon struc
            _bp          dw    ?
             _ip         dw    ?
IF COD_MARE
            _cs          dw    ?
ENDIF
IF DATE_MICI
            adr_var      dw ?
ELSE
            adr_var      dd ?
ENDIF
      sablon ends

      portabila proc
      push  bp
      mov   bp, sp
IF DATE_MARI
      les   bx, adr_var
ELSE
      lea   bx, adr_var
ENDIF
      pop   bp
      ret
      portabila endp
```

Șablonul de acces se modifică atât la schimbarea tipului datelor, cât și la cea a codului. Similar, accesul la adr_var presupune încărcarea offset-ului sau a perechii (segment:offset).

Segmentele definite care au importanță din punct de vedere ASM sunt următoarele:

_TEXT sau SURSA_TEXT

Corespund directivei **.code** din ASM sau codului generat de compilatorul C. La modelele de cod mare (compact, large, huge) se generează numele SURSA_TEXT, unde SURSA este numele fișierului sursa C sau ASM. La ASM se poate folosi directiva **.code nume**, caz în care se va genera un segment cu numele nume_TEXT. La modelele de cod mic (**tiny**, **small**, **compact**), se generează segmente cu numele _TEXT, cu tipul de combinare PUBLIC, ceea ce va face ca toate segmentele _TEXT să se combine într-unul singur, rezultând astfel un unic segment de cod.

_DATA sau SURSA_DATA

Corespund directivei **.data** din ASM sau datelor din C în clasa extern și static internal (deci cele cu alocare statică) inițializate explicit. La modelul **huge**, compilatorul C generează numele SURSA_DATA, unde SURSA este numele fișierului sursă, iar la celelalte, numele _DATA. Tipul de combinare este PUBLIC, deci toate segmentele cu numele _DATA se vor combina într-unul singur.

_BSS

Corespunde directivei **.data?** din ASM sau datelor cu alocare statică neinițializate explicit. Acestea vor fi inițializate de modulul **c0** cu valoarea 0. La modelul **huge**, atât datele neinițializate, cât și cele inițializate sunt definite în segmentele SURSA_DATA. Tipul de combinare este PUBLIC, deci toate segmentele cu numele _BSS se vor combina într-unul singur.

_STACK

Corespunde stivei programului. Stiva este declarată cu o lungime minimă în c0, dar poate fi extinsă în ASM prin directiva **.stack**, sau în C prin declarația:

```
extern unsigned _stklen = ... ;
```

La modelele diferite de **huge** și **tiny**, se generează un grup de segmente numit DGROUP, care cuprinde segmentele _DATA și _BSS (între altele). La modelul **tiny**, acest grup cuprinde și segmentele _TEXT și _STACK. La modelul **huge**, DGROUP nu cuprinde segmentele SURSA_DATA.

Compilatorul C generează următoarele segmente, funcție de modelul de memorie folosit:

- **tiny**

Se va genera un unic segment de cod, cu numele _TEXT, un unic segment de date inițializate _DATA și un unic segment de date neinițializate _BSS. Se generează un grup de segmente, numit DGROUP, care va include segmentele _TEXT, _DATA și _BSS. Nu se definește segment de stivă. Acest model corespunde programelor de tip .COM din DOS. În ASM, se vor folosi directivele **.code** pentru cod, **.data** pentru date inițializate și **.data?** pentru date neinițializate. Registrul DS este inițializat cu DGROUP. Toți pointerii sunt de tip **near**. Apelurile de funcții sunt implicit near. Modulul de inițializare este **c0t.obj**, iar bibliotecile specifice sunt **cs.lib** și **maths.lib**.

- **small**

Se va genera un unic segment de cod, cu numele _TEXT și un grup de segmente, numit DGROUP, care va include segmentul de date inițializate _DATA și segmentul de date neinițializate _BSS. Se definește explicit un segment de stivă. Toți pointerii sunt implicit de tip **near**. Apelurile de funcții sunt implicit de tip **near**. Registrul DS este inițializat în modulul c0 cu DGROUP. Modulul de inițializare specific este **c0s.obj**, iar bibliotecile specifice sunt **cs.lib** și **maths.lib**.

- **compact**

Se va genera un unic segment de cod, cu numele _TEXT și un grup de segmente de date, numit DGROUP care conține segmentele _DATA (accesibil în ASM prin directiva **.data**) și _BSS (accesibil în ASM prin **.data?**). Pointerii la funcții sunt implicit de tip **near**, iar pointerii la date sunt implicit de tip **far**.

Apelurile de funcții sunt implicit de tip **near**. Registrul DS este inițializat în modulul **c0** cu DGROUP. Modulul de inițializare specific este **c0c.obj**, iar bibliotecile specifice sunt **cc.lib** și **mathc.lib**.

- **medium**

Se vor genera mai multe segmente de cod, cu numele SURSA_TEXT, unde SURSA este numele fișierului sursă C sau ASM. În ASM, dacă directiva **.code** are un parametru, se va genera un segment de cod cu numele respectiv. Se consideră că în cuprinsul segmentului de cod respectiv, este activă o directivă ASSUME CS: cu numele segmentului respectiv. Se generează grupul DGROUP care include segmentele _DATA și _BSS (accesibile în ASM prin **.data** și **.data?**). Pointerii la date sunt implicit de tip **near**, iar pointerii la funcții sunt implicit de tip **far**. Apelurile de funcții sunt implicit de tip **far**. Registrul DS este inițializat în modulul **c0** cu DGROUP. Modulul de inițializare specific este **c0m.obj**, iar bibliotecile specifice sunt **cm.lib** și **mathm.lib**.

- **large**

Se vor genera mai multe segmente de cod, cu numele SURSA_TEXT, unde SURSA este numele fișierului sursă C sau ASM. În ASM, dacă directiva **.code** are un parametru, se va genera un segment de cod cu numele respectiv. Se consideră că, în cuprinsul segmentului de cod respectiv, este activă o directivă ASSUME CS: cu numele segmentului respectiv. Se generează grupul DGROUP care include segmentele _DATA și _BSS (accesibile în ASM prin **.data** și **.data?**). Toți pointerii sunt implicit de tip **far**. Apelurile de funcții sunt implicit de tip **far**. Registrul DS este inițializat în modulul **c0** cu DGROUP. Modulul de inițializare specific este **c0l.obj**, iar bibliotecile specifice sunt **cl.lib** și **mathl.lib**.

- **huge**

Se vor genera mai multe segmente de cod, cu numele SURSA_TEXT, unde SURSA este numele fișierului sursă C sau ASM. În ASM, dacă directiva **.code** are un parametru, se va genera un segment de cod cu numele respectiv. Se vor genera mai multe segmente de date, cu numele SURSA_DATA, unde SURSA este numele fișierului sursă C sau ASM. Nu se generează segmente _BSS, toate datele fiind definite în segmentele SURSA_DATA. Se consideră că în cuprinsul modulului respectiv este activă o directivă ASSUME CS:SURSA_TEXT, DS:SURSA_DATA. Grupul de segmente DGROUP nu include segmentele SURSA_DATA. Asamblorul TASM generează însă un segment cu numele _DATA, corespunzător directivei **.data**, care se include în grupul DGROUP. Registrul DS este inițializat în modulul **c0** cu DGROUP, iar la fiecare intrare într-o funcție C (inclusiv **main**), registrul DS este inițializat cu SURSA_DATA. Evident, la intrarea în funcțiile ASM, va trebui făcută o inițializare cu DGROUP, salvând în prealabil registrul DS în stivă și refăcându-l la ieșirea din funcția ASM. Pentru accesul la segmentele de date definite în C se

poate folosi operatorul SEG aplicat unei variabile externe sau direct numele segmentului, în forma SURSA_DATA.

Toți pointerii sunt implicit de tip **far** și sunt normalizați, adică offset-ul este redus la intervalul de valori 0...15. Astfel, corespondența **segment:offset** cu adresa fizică de memorie devine biunivocă. Incrementarea/decrementarea pointerilor se va reflecta acum atât în offset, cât și în adresa de segment, spre deosebire de modelele celelalte, la care incrementarea/decrementarea pointerilor de tip far acționează numai asupra offset-ului. Este posibilă deci definirea unor structuri de date de dimensiuni mai mari decât 64K, ceea ce la celelalte modele de memorie nu este posibil.

Apelurile de funcții sunt implicit de tip far. Modulul de inițializare specific este **c0h.obj**, iar bibliotecile specifice sunt **ch.lib** și **mathh.lib**.

O problemă specială poate apărea la modelele de memorie așa-zise „de date mici" (**small** și **medium**). Se pot defini date de orice fel, atât în clasa de alocare automatic (cu spațiu rezervat în stivă, deci accesibile prin registrul de segment SS), cât și în clasa de alocare static (cu spațiu rezervat într-unul din segmentele ce compun grupul DGROUP, deci accesibile prin registrul de segment DS).

În aceste modele de memorie, pointerii către date sunt de tip **near**, deci ei memorează numai deplasamentele obiectelor în cadrul segmentelor respective (de date, pentru obiecte în clasa static, respectiv de stivă, pentru obiecte în clasa auto). Dacă se accesează obiecte (date) prin pointeri, nu se poate ști în ce tip de segmente sunt definite aceste obiecte. Problema este rezolvată (în cazul modelelor **small** și **medium**) prin generarea segmentelor de așa manieră, încât adresele de început ale segmentului de stivă și ale grupului DGROUP să coincidă. Practic, pe parcursul execuției, registrele DS și SS vor avea aceeași valoare.

Să observăm că problema nu se pune în cazul modelului **tiny** (există un unic segment) și nici în cazul modelelor **large** și **huge** (toți pointerii sunt de tip **far**). De asemenea, problema nu se pune în cazul modelului „de cod mic și date mari" **compact**, deoarece pointerii către funcții vor conține (indiferent de locul unde sunt definiți) deplasamente în cadrul unicului segment de cod care se generează. Apelurile de funcții prin pointeri se traduc în simple salturi indirecte intrasegment (instrucțiuni de tip **call word ptr** ...). Exemple de asemenea apeluri sunt:

```
call  word ptr [bp-4]          ; Apel indirect, pointer
                               ; in clasa auto
call  word ptr _p              ; Apel indirect, pointer
                               ; in clasa static
```

Dacă se fac indirectări multiple, acestea presupun definirea unor pointeri către pointeri către funcții, adică a unor pointeri către date. Acești pointeri vor fi deci

de tip **far** și vor fi încărcați și dereferențiați totdeauna ca adrese complete (segment și offset).

În concluzie, să reținem că, la modelele small și medium, registrele DS și SS vor avea același conținut la execuție. Mediul integrat oferă opțiunea **Options/Compiler/Code Generation/Assume DS not equal SS**, care este recomandabil să fie plasată pe **Default for memory model**.

Pe lângă segmentele descrise mai sus, modulul de inițializare **c0** mai definește o serie de segmente, folosite la inițializare, la ieșirea în sistemul de operare etc, dar care nu prezintă interes din punct de vedere ASM. Se poate vedea exact ce segmente se generează, studiind fișierul sursă **c0.asm**, furnizat în kit-ul implementării C (în directorul **\examples\startup**).

Când există îndoieli asupra a ceea ce se generează concret de către compilatorul C, se poate folosi opțiunea -S la compilare în linia de comandă, obținându-se în acest fel un fișier sursă ASM, corespunzător modulului C respectiv sau se poate lansa Turbo Profiler-ul din mediu integrat, care, între alte informații, furnizează și codul ASM generat.

7.4 O aplicație mixtă cu tipuri de date simple

Pentru a fixa ideile, considerăm un exemplu de program, compus dintr-un modul C și un modul ASM, care va fi detaliat în trei cazuri de modele: compact, medium și huge. Codul ASM pentru celelalte modele se deduce simplu: la modelul small se tratează datele ca la medium și codul ca la compact, iar la modelul large se tratează datele ca la compact și codul ca la medium.

Modulul sursă C, presupus în fișierul test1c.c, este același în toate situațiile, și anume:

```
#include <stdio.h>
extern int f (int);
extern int (*pf) (int);
extern int *g (void);
extern int *gg (void);
extern int ext_i;
extern int ext_n;
int extc_i = 2;

int *ff (int *pi)
{
       *pi *= 2;
      return pi;
}
void main(void)
{
      int n;
      printf("\nIntroduceti un intreg: ");
      scanf("%d",&n);
      n = f(n);
```

```
        printf("Valoarea dubla este: %d\n",n);
        n = (*pf)(n);
        printf("Valoarea dubla este: %d\n",n);
        printf("Variabila externa initializata este: %d\n", ext_i);
        ext_n = 5;
        printf("Variabila externa neinitializata este: %d\n", *g());
        printf("Valoarea dubla este: %d\n", *gg());
}
```

Modulele ASM sunt presupuse în fișierele test1ca.asm, test1ma.asm și, respectiv, test1ha.asm, corespunzătoare modelelor compact, medium și huge.

În aplicația studiată, se consideră următoarele entități de program definite în ASM:

- funcția f, cu tipul int și cu un parametru de tip int;
- pointerul la funcție pf, inițializat cu adresa funcției f;
- funcția g, cu tipul int și fără parametri;
- funcția gg, cu tipul int * și fără parametri;
- întregul ext_i, inițializat cu valoarea 11;
- întregul ext_n, neinițializat.

În modulul C, se consideră următoarele entități:

- întregul extc_i, inițializat cu valoarea 2;
- funcția ff, cu tipul int * și cu un parametru de tip int *.

Funcția ff primește o adresă de întreg și înmulțește cu 2 întregul de la adresa respectivă, întorcând adresa primită.

Funcția f primește un întreg, afișează la consolă mesajul „Întregul este:" și valoarea variabilei extc_i. Se întoarce programului apelant valoarea dublă a parametrului primit.

Funcția g întoarce programului apelant adresa variabilei ext_n.

Funcția gg apelează funcția ff, cu parametrul adresa variabilei ext_n, întorcând programului apelant adresa returnată de funcția ff, adică adresa aceleiași variabile ext_n.

Programul principal, scris în C, citește de la consolă un întreg n și apelează de două ori funcția f, cu parametrul n, prima dată direct, iar a doua oară prin pointerul pf. În ambele apeluri, valoarea întoarsă este depusă în aceeași variabilă n. Se afișează apoi variabila ext_i, se modifică explicit variabila ext_n și se apelează funcțiile g și gg, afișând la întregii de la adresele întoarse de aceste funcții, adică variabila ext_n.

Afișările din modulul ASM sunt realizate prin macroinstrucțiunile putsi și puti, care presupun datele adresabile prin registrul DS. Rezultă, deci, că aplicația va trebui să cuprindă și modulul obiect io.obj (vezi Anexa B).

7.4.1 Dezvoltare în modelul de memorie compact

Modulul test1ca.asm, corespunzător modelului compact, este:

```
.model compact
include io.h
public        _f,  _pf,   _g, _gg, _ext_i, _ext_n
extrn         _ff: near,  _extc_i: word
sablon _f struc
        _bp   dw ?                ; Sablon pentru accesul
        _ip   dw ?                ; in stiva, in procedura
        n     dw ?                ; de tip near f
sablon_f ends
.data
      _ext_i        dw     11     ; intreg initializat
      _pf           dw     _f     ; Pointer near initializat cu
                                  ; adresa (offset-ul) lui _f
.data?
      _ext_n        dw     ?      ; intreg neinitializat
.code
_f  proc near
      push  bp
      mov   bp, sp                ; Pozitionare bp pentru acces
      mov   ax, [bp].n            ; Preluare parametru
      add   ax, ax                ; Dublare
      putsi <cr,lf,'Intregul este: '>
      mov   bx, DGROUP:_extc_i          ; Segmentul este DGROUP
      puti  bx                          ; Tipareste
                                        ; intregul (bx)
      putsi <cr, lf>
      pop   bp
      ret                               ; (ax) = rezultat
_f  endp

_g  proc near
      mov   dx, DGROUP                  ; Segment
      lea   ax, DGROUP:_ext_n           ; Offset
      ret                               ; Rezultat in (dx:ax)
_g endp

_gg proc near
      mov   ax, DGROUP                  ; Pregatire parametri
                                        ; apel
      push  ax                          ; Segment
      lea   ax,DGROUP:_ext_n
      push  ax                          ; Offset
      call  near ptr _ff                ; Apel _ff
      add   sp, 4                       ; Descarcare stiva
      ret                               ; Rezultat in (dx:ax)
_gg endp
end
```

Deoarece modelul de memorie este compact, toate apelurile de funcții sunt de tip near, pointerul pf la funcție este de tip near, iar toți pointerii la date sunt de tip

far. Funcțiile f, g și gg vehiculează adrese de 32 de biți, transmise prin stivă sau întoarse prin perechea (DX:AX).

Programul va afișa la consolă textul:

```
Introduceti un intreg: 13
Intregul C este: 2
Valoarea dubla este: 26
Intregul C este: 2
Valoarea dubla este: 52
Variabila externa initializata este: 11
Variabila externa neinitializata este: 5
Valoarea dubla este: 10
```

Presupunând directorul mediului C ca fiind C:\BC, secvența de linii de comandă pentru dezvoltarea aplicației este:

```
bcc -mc -c -Ic:\bc\include test1c.c
tasm test1ca.asm/ml
tlink c:\bc\lib\c0c test1c test1ca io,test1c,,c:\bc\lib\cc
```

7.4.2 Dezvoltare în modelul de memorie medium

Modulul test1ma.asm, corespunzător modelului de memorie medium, este:

```
.model medium
include io.h
public        _f, _pf, _g, _gg, _ext_i, _ext_n
extrn       _ff: far, _extc_i :word
sablon_f struc
      _bp          dw ?          ; Sablon pentru accesul
      _ip_cs       dd ?          ; in stiva, in procedura
      n            dw ?          ; de tip far _f
sablon_f ends
.data
      _ext_i       dw 11         ; intreg initializat
      _pf          dd _f         ; Pointer far initializat cu
                                 ; adresa completa a lui _f
.data?
      _ext_n       dw ?          ; intreg neinitializat
.code
_f    proc far
      push  bp
      mov   bp, sp               ; Pozitionare bp pentru acces
      mov   ax, [bp].n           ; Preluare parametru
      add   ax, ax               ; Dublare
      mov   bx, DGROUP:_extc_i; Segmentul este DGROUP
      putsi <cr, lf,'Intregul C este: '>
      puti  bx                   ; Tipareste intregul (bx)
      putsi <cr, lf>
      pop   bp
      retf                       ; (ax) = rezultat
_f    endp
```

```
_g    proc far
      lea   ax,DGROUP:_ext_n  ; Offset
      retf                    ; Rezultat in (ax)
_g endp

_gg   proc far
      lea   ax, DGROUP:_ext_n ; Pregatire parametri apel
      push  ax                ; Offset
      call  far ptr _ff       ; Apel _ff
      add   sp, 2             ; Descarcare stiva
      retf                    ; Rezultat in (ax)
_gg   endp
end
```

Ceea ce se schimbă față de modelul anterior este faptul că apelurile și pointerii la funcții sunt de tip far, iar adresele și pointerii la variabile sunt de tip near. Șablonul de acces în stivă se modifică, ținând cont că un apel far salvează și registrul CS în stivă. Pointerul pf se definește cu directiva dd (define doubleword). Adresa lui ext_n (offset) este întoarsă de funcțiile g și gg prin AX.

Secvența de linii de comandă este:

```
bcc -mm -c -Ic:\bc\include test1c.c
tasm test1ma.asm/ml
tlink c:\bc\lib\c0m test1c test1ma io,test1m,,c:\bc\lib\cm
```

7.4.3 Dezvoltare în modelul de memorie huge

Modulul test1ha.asm, corespunzător modelului huge, este:

```
.model huge
include io.h
public _f, _pf, _g, _gg, _ext_i, _ext_n
extrn _ff:far, _extc_i:word

sablon_f struc
      _bp         dw ?        ; Sablon pentru accesul
      _ip_cs      dd ?        ; in stiva, in procedura
      n           dw ?        ; de tip far _f
sablon_f ends

.data
      _ext_i      dw 11       ; intreg initializat
      _pf         dd _f       ; Pointer far initializat cu
                              ; adresa completa a lui _f
.data?
      _ext_n      dw ?        ; intreg neinitializat

 .code
      _f proc     far
      push        bp
      mov         bp,sp       ; Pozitionare bp pentru acces
      push        ds          ; Salvare ds
      push        es          ; Salvare es
      push        bx          ; Salvare bx
```

```
        mov     ax,DGROUP
        mov     ds,ax                   ; Segment local de date

        mov     ax,[bp].n               ; Preluare parametru
        add     ax,ax                   ; Dublare

        push    ax                      ; Salvare rezultat
        mov     ax,SEG _extc_i          ; Seg. asociat lui _extc_i
        mov     es,ax
        mov     bx,es:_extc_i           ; Preluare _extc_i
;
; Sirul din macroinstructiunea putsi se defineste intr-o
; directiva .data, deci in segmentul local, de aceea ds
; trebuie sa indice DGROUP
;
        putsi <cr,lf,'Intregul C este: '>
        puti  bx                        ; Tipareste intregul (bx)
        putsi <cr,lf>
        pop   ax                        ; Rezultat

        pop   bx                        ; Refacere bx
        pop   es                        ; Refacere es
        pop   ds                        ; Refacere ds

        pop bp
        retf                            ;(ax) = rezultat
        _f endp

_g proc far
         mov   dx,DGROUP                ; Segment
        lea    ax,DGROUP:_ext_n         ; Offset
        retf                            ;Rezultat in (dx:ax)
_g endp

_gg proc far
        mov    ax,DGROUP                ; Pregatire parametri
        push   ax                       ; Segment local
        lea    ax,DGROUP:_ext_n
        push   ax                       ; Offset
        call   far ptr _ff              ; Apel _ff
        add    sp,4                     ; Descarcare stiva
        retf                            ; Rezultat in (dx:ax)
_gg endp
end
```

Se observă că, în procedura f, unde se face acces la date, se poziționează explicit registrul DS pe grupul de segment DGROUP, pentru a putea utiliza macroinstrucțiunile de afișare și se încarcă explicit în ES adresa de segment a variabilei externe ext_c. Registrele DS și ES se salvează la intrarea în funcție și se refac înainte de întoarcerea în programul apelant.

Secvența de linii de comandă este:

```
bcc -mh -c -Ic:\bc\include test1c.c
```

```
tasm test1ha.asm/ml
tlink c:\bc\lib\c0h test1c test1ha io,test1h,,c:\bc\lib\ch
```

7.5 O aplicație mixtă cu tipuri de date structurate

Se consideră modulul sursă C, presupus în fișierul test2c.c:

```
#include <string.h>
struct tips { int n; char *sir; };
void f (struct tips *);
struct tips *g (struct tips *);
struct tips * (*pg) (struct tips *) = g;

struct tips a = { 100, "Iata un sir..." };

struct tips *g(struct tips *x)
{
      static int contor = 0;
      contor++;
      f(x);
      (x -> n) ++;
      if(contor % 2)
            strupr(x -> sir);
      else
            strlwr(x -> sir);
      f(x);
      return x;
}
```

Se consideră tipul de structură tips, care cuprinde un întreg și un pointer la un șir de caractere. Structura a este inițializată cu valoarea 100, respectiv cu șirul „Iata un sir...".

Funcția de tip void f, scrisă în ASM, primește o adresă de structură și afișează la consolă cele două câmpuri (întreg și șir de caractere) cu ajutorul macroinstrucțiunilor puti și puts, definite în fișierul header io.h.

Funcția g, scrisă în C, primește adresa x a unei structuri și realizează următoarele operații:

- apelul funcției f, cu parametrul adresa x (care va afișa structura indicată de x);
- incrementarea câmpului numeric al structurii de la adresa x;
- modificarea succesivă a șirului de caractere din structura indicată de variabila x, din litere mici în litere mari și succesiv; acest lucru este realizat prin incrementarea modulo 2 a unei variabile locale statice;
- apelul funcției f, cu parametrul adresa x (care va afișa structura indicată de x).

Programul principal, scris în ASM, realizează următoarele operații:

- apelează funcția g în mod direct, cu parametrul adresa structurii a;

- pregătește în stivă adresa întoarsă de g (deci adresa structurii a), pentru un apel viitor;
- apelează funcția f în mod direct, cu parametrul întors de apelul lui g (deci cu adresa structurii a);
- apelează indirect funcția g, prin intermediul pointerului pg, cu parametrul adresa pregătită anterior în stivă;
- apelează funcția f, cu parametrul întors de apelul indirect al lui g (deci cu adresa structurii a).

7.5.1 Dezvoltare în modelul de memorie compact

Modulul test2ca.asm, corespunzător modelului de memorie compact, este:

```
.model      compact
include     io.h
public      _f, _main
extrn       _g: near, _a: far, _pg: word

tips  struc
      n     dw ?                ; Sablon pentru
      sir   dd ?                ; structura tips
tips  ends

sablon struc
      _bp          dw ?         ; Sablon pentru acces
       _ip         dw ?         ; in stiva
      _adr_struc   dd ?         ; Parametrul
sablon ends

.stack      1024
.code

_f    proc near
      push  bp
      mov   bp, sp
      push  ds                  ; Salvari
      push  di                  ; registre
      push  si                  ; folosite

;
; incarcare adresa far din stiva
;
      les   di, dword ptr [bp]._adr_struc
;
; Acces la membrul n al structurii
;
      mov   ax, es:[di].n
;
; Tiparire n
;
      putsi       <cr, lf, 'Intregul este: '>
      puti  ax
      putsi       <cr, lf, 'Sirul este: '>
```

```
; Acces la membrul sir (pointer far) al structurii
;
      lds   si, dword ptr es:[di].sir
;
; Tiparire sir
;
      call  far ptr puts_proc
      pop   si                  ; Refaceri
      pop   di                  ; registre
      pop   ds                  ; folosite
      pop   bp
      ret
_f endp

_main proc near
      mov   ax,DGROUP           ; Pregatire parametri
      push  ax                  ; Segmentul structurii a
      lea   ax,DGROUP:_a
      push  ax                  ; Offset-ul structurii a
      call  near ptr _g         ; Apel g
      add   sp,4                ; Descarcare stiva
;
; g intoarce un pointer far in (dx:ax)
; Este pregatit acest pointer in stiva,
; pentru un apel viitor al lui g
; (altfel ar fi trebuit salvat intr-o zona de date)
;
      push  dx                  ; Segment
      push  ax                  ; Offset

;
; Pregatire parametru pentru apelul lui f
;
      push  dx                  ; Segment
      push  ax                  ; Offset
      call  near ptr _f         ; Apel f
      add   sp, 4               ; Descarcare stiva
;
; Acum se face apelul lui g, indirect, prin pointerul
; de tip near pg, cu parametrul de tip pointer far,
; pregatit anterior in stiva
;
      call  word ptr _pg
      add   sp, 4               ; Descarcare stiva
;
; g intoarce un pointer far in (dx:ax)
; Se pregateste acest pointer in stiva, pentru apelul lui f
;
      push  dx                  ; Segment
      push  ax                  ; Offset
      call  near ptr _f         ; Apel f
      add   sp, 4               ; Descarcare stiva
      ret                       ; Terminare program
_main endp
end
```

Pointerii la date sunt de tip far, iar pointerul lka funcție este de tip near. Se observă declararea simbolului extern a ca o etichetă de tip far, ceea ce este în concordanță cu convenția că numele unei variabile este sinonim cu adresa variabilei respective.

La consolă se va tipări:

```
Intregul este: 100
Sirul este: Iata un sir...
Intregul este: 101
Sirul este: IATA UN SIR...
Intregul este: 101
Sirul este: IATA UN SIR...
Intregul este: 101
Sirul este: IATA UN SIR...
Intregul este: 102
Sirul este: iata un sir...
Intregul este: 102
Sirul este: iata un sir...
```

Secvența de linii de comandă este:

```
bcc -mc -c -Ic:\bc\include test2c.c
tasm test2ca.asm/ml
tlink c:\bc\lib\c0c test2ca test2c io,test2c,,c:\bc\lib\cc
```

7.5.2 Dezvoltare în modelul de memorie medium

Modulul test2ma.asm, corespunzător modelului medium, este:

```
 .model medium
 include io.h
 public _f, _main
 extrn _g:far, _a:near, _pg:dword
 tips struc
      n            dw ?              ; Sablon pentru
      sir          dw ?              ; structura tips
 tips ends

 sablon struc
      _bp          dw ?              ; Sablon pentru acces
      _ip_cs       dd ?              ; in stiva
      _adr_struc   dw ?              ; Parametrul
 sablon ends

 .stack     1024
.code

 _f proc far
      push         bp
      mov          bp,sp
      push         di                ; Registre
      push         si                ; folosite
```

```
;
        mov         di,word ptr [bp]._adr_struc
;
        mov         ax,DGROUP:[di].n
;
        putsi       <cr,lf,'Intregul este: '>
        puti        ax
        putsi       <cr,lf,'Sirul este: '>
;
        mov         si,word ptr DGROUP:[di].sir
;
        call        far ptr puts_proc
        pop         si                  ; Refaceri
        pop         di                  ; registre
        pop         bp
        retf
 _f endp

 _main proc far
        lea         ax,DGROUP:_a        ; Pregatire parametru
        push        ax                  ; Offset-ul structurii a
        call        far ptr _g          ; Apel g
        add         sp,2                ; Descarcare stiva

;
; g intoarce un pointer near în (ax)
;
        push        ax                  ; Offset
;
; Pregatire parametru pentru apelul lui f
;
        push        ax                  ; Offset
        call        far ptr _f          ; Apel f
        add         sp,2                ; Descarcare stiva
;
        call        dword ptr _pg
        add         sp,2                ; Descarcare stiva
;
; g intoarce un pointer near in (ax)
;
        push        ax                  ;Offset
        call        far ptr _f          ; Apel f
        add sp,2                        ; Descarcare stiva
        retf                            ; Terminare program
 _main endp
 end
```

Diferențele față de cazul precedent rezultă din faptul că funcțiile și pointerii la funcții sunt de tip far, iar adresele variabilelor și pointerii la date sunt de tip near.

Simbolul extern a (de tip structură) se declară în ASM ca o etichetă de tip near.

Secvența de linii de comandă este:

```
bcc -mm -c -Ic:\bc\include test2c.c
tasm test2ma.asm/ml
```

```
tlink c:\bc\lib\c0m test2ma test2c io,test2m,,c:\bc\lib\cm
```

7.5.3 Dezvoltare în modelul de memorie huge

Modulul test2ha.asm, corespunzător modelului huge este:

```
.model huge
 public _f, _main
 extrn _g:far, _a: far, _pg:dword
 include io.h

 tips struc
      n           dw ?              ; Sablon pentru
      sir         dd ?              ; structura tips
 tips ends

 sablon struc
      _bp         dw ?              ; Sablon pentru acces
      _ip_cs      dd ?              ; in stiva
      _adr_struc  dd ?              ; Parametrul
 sablon ends

.stack      1024
.code

 _f proc far
      push        bp
      mov         bp,sp
      push        ds                ;Salvari
      push        di                ;registre
      push        si                ;folosite
;
      mov         ax,DGROUP
      mov         ds,ax
;
      les         di,dword ptr [bp+6]
      mov         ax,es:[di].n
      putsi       <cr,lf,'Intregul este: '>
      puti        ax
      putsi       <cr,lf,'Sirul este: '>
      lds         si,dword ptr es:[di].sir
      call        far ptr puts_proc
;
      pop         si                ; Refaceri
      pop         di                ; registre
      pop         ds                ; folosite
      pop         bp
      retf
 _f endp

 _main proc far
      push        ds
      push        es
      mov         ax,DGROUP         ; Comutare pe segmentul
      mov         ds,ax             ; propriu de date
```

```
;
        mov         ax,SEG _a           ; Pregatire parametru
        mov         es,ax               ; Segmentul structurii a
        push        es
        lea         ax,es:_a            ; Offset-ul structurii a
        push        ax
        call        far ptr _g          ; Apel g
        add sp,4                        ; Descarcare stiva
;
;       g intoarce un pointer far in (dx:ax)
;       Este pregatit acest pointer in stiva, pentru
;       un apel viitor al lui g (altfel ar fi trebuit salvat
;       intr-o zona de date)
;
        push dx                         ; Segment
        push ax                         ; Offset
;
;       Pregatire parametru pentru apelul lui f
;
        push        dx                  ; Segment
        push        ax                  ; Offset
        call        far ptr _f          ; Apel f
        add         sp,4                ; Descarcare stiva
;
; Acum se face apelul lui g, indirect, prin pointerul
; de tip near pg (care e in alt segment de date), cu
; parametrul de tip pointer far, pregatit anterior
; in stiva
        mov         ax, SEG _pg
        mov         es,ax
        call        dword ptr es:_pg
        add         sp,4
;
; g intoarce un pointer far in (dx:ax)
; Se pregateste acest pointer in stiva, pentru apelul lui f
;
        push        dx
        push        ax
        call        far ptr _f
        add         sp,4
;
        pop es
        pop ds
        retf
 _main endp
 end
```

Diferența față de celelalte cazuri este că datele externe nu mai sunt în mod necesar definite în același segment (grup) cu datele din modulul ASM. Adresarea structurii a și a pointerului pg trebuie făcută în consecință, folosind operatorul SEG pentru luarea segmentelor în care aceste simboluri sunt definite.

Secvența de linii de comandă este:

```
bcc -mh -c -Ic:\bc\include test2c.c
tasm test2ha.asm/ml
tlink c:\bc\lib\c0h test2ha test2c io,test2h,,c:\bc\lib\ch
```

7.6 Dezvoltarea într-un limbaj mixt (C și ASM)

Mediul de dezvoltare Borland C oferă posibilitatea inserării de cod ASM în programe sursă C. Fiecare instrucțiune ASM trebuie precedată de cuvântul cheie asm sau să fie inclusă într-un bloc de tip ASM, de forma:

```
asm {
        ; Instructiuni ASM
}
```

Acest model de dezvoltare este foarte comod, deoarece vizibilitatea obiectelor definite în C se extinde și asupra blocurilor ASM. Astfel, putem folosi nume de variabile, nume de parametri formali ai funcțiilor etc., fără a mai scrie caracterul _ în fața numelor respective. Se evită astfel și secvențele de intrare și revenire din funcții, care sunt realizate automat de compilatorul C.

O altă facilitate este accesul direct din C asupra registrelor procesorului, realizat prin **variabilele C predefinite _AX, _BX, _CX** etc., care pot fi utilizate, de exemplu, în instrucțiuni de atribuire.

Există restricții de utilizare asupra etichetelor. Mai precis, etichetele trebuie definite ca etichete C, deci în afara blocurilor ASM.

Ca și la dezvoltarea în module separate, programatorul trebuie să gestioneze explicit modelele de memorie, în sensul definirii precise a adreselor far sau near. Pentru aceasta, se utilizează modelele de memorie C și/sau cuvintele cheie near și far ale implementării Borland C.

Un modul dezvoltat în această manieră este scris de fapt într-un limbaj mixt (C și ASM). De altfel, fișierele sursă de acest tip se creează de obicei cu extensia .CAS (de la C și ASM).

Pentru exemplificare, să considerăm o funcție de căutare liniară într-un tablou de întregi. În limbajul C, am scrie această funcție în forma:

```
int cauta (int *a, size_t n, int x)
{
      int i;
      for (i = 0; i < n; i++)
            if (x == a[i])
                  return i;    /* S-a gasit intregul x */
      return -1;               /* Nu s-a gasit intregul x */
}
```

Dorim acum să scriem aceeași funcție în „limbajul CAS". Presupunem pentru început un model „de date mici". Implementarea funcției este următoarea:

```
int cauta (int near *a, size_t n, int x)
{
      asm {
            push  si                  /* Salvari */
            push  cx                  /* registre */
            push  dx

            mov   si, a               */ Adresa near */
                                      */ a tabloului */
            mov   cx, n               */ Numar elemente */
            jcxz  not_gasit           */ Test n == 0 */
            mov   dx, x               */ Element cautat */
            sub   si, 2               */ Pornim cu o pozitie */
                                      */ mai la stanga */
      }
reluare:
      asm {
            add   si, 2               */ Element curent */
                                      */ din tablou */
            cmp   [si], dx            */ Comparam cu */
                                      */ cel cautat */
            loopnz reluare            */ Nu este, se reia */
                                      */ bucla */
            jnz   not_gasit           */ Am iesit din bucla */
            mov   ax, n               */ Daca ZF a fost 1, */
                                      */ calculam */
            sub   ax, cx              */ indicele elementului */
                                      */ gasit */
            dec   ax                  */ ca n - CX - 1 */
            jmp   final               */ Salt la iesire */
      }
not_gasit:
      asm   mov   ax, -1              */ Nu s-a gasit */
                                      */ elementul x */
final:
      asm {
            pop   dx                  */ Refaceri */
            pop   cx                  */ registre */
            pop   si
      }
      return      _AX                 */ Instructiune */
}
```

Se observă tehnica de scriere a buclei de căutare (prefixul LOOPNZ) și calculul indicelui elementului găsit, ca diferență dintre dimensiunea n a tabloului și valoarea incrementată a contorului CX la ieșirea din buclă. Revenirea în funcția apelantă se face cu instrucțiunea C return _AX (ne amintim convenția de întoarcere a datelor simple).

Un punct important este încărcarea adresei tabloului. Dacă adresa este near, o încărcăm cu o instrucțiune MOV (în stivă se găsește deplasamentul în cadrul segmentului curent de date). În cazul unui model de „date mari", în stivă se

găsește un pointer far (o variabilă de tip DoubleWord), caz în care încărcarea s-ar face cu o instrucțiune:

```
les   si, a                 */ Adresa far a tabloului */
```

Compararea elementului curent cu cel căutat s-ar scrie:

```
cmp   es:[si], dx           */ Comparam cu cel cautat */
```

În fine, ar mai trebui salvat și restaurat registrul ES, iar prototipul funcției ar trebui scris în forma:

```
int cauta (int far *a, size_t n, int x);
```

Modificările de mai sus transformă funcția într-una adecvată modelelor „de date mari".

Un program de test pentru funcția de mai sus s-ar scrie în genul:

```
#include <stdio.h>
int x [ ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, };
#define NREL(a) (sizeof(a)/sizeof(a[0]))
void main (void)
{
      int i, val = 6;
      i = cauta (x, NREL(x), val);
      if (i > 0)
            printf("Elementul %d se afla pe pozitia %d din
                  tablou\n", val, i);
      else
            printf("Elementul %d nu se afla in tablou\n", val);
}
```

Trebuie totuși observat că această tehnică mixtă de dezvoltare are și dezavantaje. Un modul CAS este destul de greu de întreținut, iar transportul său în alt mediu de dezvoltare decât Borland ar putea crea probleme de compatibilitate. Tehnica mixtă merită folosită atunci când avem de implementat secvențe relativ scurte de program, pentru care nu se justifică un modul (fișier sursă) separat.

7.7 Performanțe în ASM și C

În acest subcapitol, vom face o analiză comparativă a unui algoritm implementat în C și ASM. Algoritmul ales este un algoritm de sortare internă, anume sortarea prin partiționare și interschimbare, numită și sortare rapidă (quicksort).

Algoritmul este recursiv și se poate implementa foarte comod atât în C, cât și în ASM. O implementare polimorfică a acestui algoritm este oferită și de biblioteca standard a limbajului C.

7.7.1 Implementarea în C

Varianta C considerată va fi de asemenea polimorfică (capabilă să sorteze tablouri de orice fel). Prototipul funcției C este:

```
void qsort (void *tab, size_t n, size_t size, PFCMP cmp);
```

Semnificația parametrilor este:

- tab - adresa de început a tabloului;
- n - numărul de elemente ale tabloului;
- size - dimensiunea în octeți a unui element;
- cmp - pointer la o funcție externă de comparație, definit prin:

```
typedef int (*PFCMP) (void *a, void *b);
```

Funcția de comparație (scrisă de utilizator) primește adresele a două elemente din tablou și întoarce o valoare negativă, zero sau pozitivă, după cum elementul de la prima adresă este „mai mare", „egal" sau „mai mic" decât elementul de la a doua adresă. Sensul noțiunilor mai mare, egal sau mai mic este complet abstract, fiind definit de utilizator chiar prin funcția de comparație.

Pentru transferul datelor, se vor utiliza funcțiile ajutătoare swap și copy, listate mai jos.

```
typedef unsigned char BYTE;
void swap(void *a, size_t size, int i, int j)
{
      /* Schimba a[i] cu a[j]. Elementele tabloului a[] au
      size octeti */
      register BYTE *b = ((BYTE *)a) + i * size;
      register BYTE *c = ((BYTE *)a) + j * size;
      register BYTE temp;
      while(size--) {
            temp = *b;
            *b++ = *c;
            *c++ = temp;
      }
}

void copy(void *a, void *b, size_t size)
{
      /* Copiaza obiectul de la adr. b in cel de la adr. a.
         Obiectele au size octeti */
      memcpy(a, b, size);
}
```

Algoritmul de sortare rapidă este recursiv. Funcția care implementează algoritmul recursiv primește ca date de intrare o parte a tabloului care trebuie sortat, prin adresa de început și doi indici notati left și right. Inițial, funcția se apelează cu indicii 0 și n-1.

Se alege un element arbitrar al tabloului v, numit pivot, notându-l cu mark (variante uzuale sunt v[left] sau v[(left+right)/2]). Se partiționează tabloul în raport cu mark, în sensul că toate elementele aflate la stânga lui mark să fie mai mici sau egale cu acesta, iar toate elementele aflate la dreapta lui mark să fie mai mari sau egale cu acesta.

În acest moment, pivotul se află pe poziția sa finală, iar tabloul este partiționat în două subtablouri. Se apelează acum aceeași funcție, cu indicii left și k-1, respectiv k+1 și right, unde k este indicele pivotului în urma partiționării. În felul acesta, se sortează cele două subtablouri. Dacă left >= right, algoritmul se oprește.

Algoritmul se poate îmbunătăți în felul următor. Să pornim cu doi indici i și j, inițializați cu left și, respectiv, cu right. Cât timp v[i] < mark, incrementăm i, apoi, cât timp v[j] > mark, decrementăm j. Dacă acum i <= j, interschimbăm v[i] cu v[j], actualizând similar indicii i și j. Tot acest proces continuă până cînd i > j.

Acum se apelează recursiv funcția, cu indicii left și j, respectiv i și right (dacă left < j, respectiv dacă i < right).

În implementarea polimorfică, deoarece la compilare nu se cunoaște dimensiunea unei înregistrări, se alocă spațiu dinamic pentru înregistrarea mark, copiindu-se elementul din mijlocul tabloului și eliberând spațiul la ieșirea din funcție.

O îmbunătățire posibilă a metodei este recurgerea la o metodă directă de sortare, dacă lungimea tabloului este inferioară unei limite prestabilite, evitând astfel apelul recursiv. De exemplu, la un tablou de 1000 de înregistrări, când dimensiunea partiției ajunge 2, se vor face 500 de apeluri ale funcției, pentru a sorta de fiecare dată un tablou de 2 elemente. Este mult mai eficient dacă acest lucru se face direct. În implementarea de față, s-a ales limita 2, caz în care cele două înregistrări se compară direct. Pentru eficiență, variabilele intens folosite în etapa de partiționare (i și j) sunt declarate în clasa register. Implementarea este următoarea:

```
void quick_sort(BYTE *v, size_t size, int left, int right,
                PFCMP cmp)
{
      register int i, j;
      BYTE *mark;

      i = left; j = right;
      switch(j - i) {
            case 0: return;
            case 1: if(cmp(v + i*size, v + j*size) > 0)
                        swap(v, size, i, j);
                    return;
            default: break;
      }
```

```
        mark = (BYTE *) malloc(size);
        copy(mark, v + ((left + right)/2)*size, size);
        do {
              while(cmp(v + i*size, mark) < 0)
                    i++;
              while(cmp(v + j*size, mark) > 0)
                    j--;
              if(i <= j)
                    swap(v, size, i++, j--);
        } while (i <= j);
        if(left < j)
              quick_sort(v, size, left, j, cmp);
        if(i < right)
              quick_sort(v, size, i, right, cmp);
        free(mark);
}

void Quick_C (void *v, size_t n, size_t size, PFCMP cmp)
{
        quick_sort(v, size, 0, (int) n-1, cmp);
}
```

7.7.2 Implementarea în ASM

Să considerăm acum implementarea algoritmului de sortare rapidă în limbaj de asamblare, într-o versiune specializată (pentru tablouri de întregi). Toate adresele se consideră de tip near, fiind offset-uri în cadrul segmentului adresat prin registrul DS.

Algoritmul este același cu cel prezentat în 7.7.1, implementarea ASM urmărind exact programul C. Variabilele din descrierea în C a algoritmului sunt menținute în registrele procesorului și în stivă.

Parametrii se transmit prin stivă, conform șablonului definit în program, iar stiva este descărcată de programul apelant. Această modalitate de apel permite ca funcția ASM să poată fi apelată din C, dacă modelul cu care se lucrează este small.

```
.286
.model small
.code
public      _qasm
;
;     qasm(int v[], int left, int right)
;
sablon      struc
      _bp_ip      dw 2 dup(?)
      v           dw ?
      left        dw ?
      right       dw ?
sablon ends
```

```
_qasm proc near
        push  bp
        mov   bp,sp
        push  a
;
;       Asignarea variabilelor
;
;       i = si, j = di, v = bx, mark = dx
;       left, right = în stivă
;
        mov   bx, [bp].v          ; v
        mov   si, [bp].left       ; i = left
        mov   di, [bp].right      ; j = right
        mov   ax, di              ; compara
        sub   ax, si              ; right - left
        cmp   ax, 1               ; cu 1
        jng   et00
        jmp   et1                 ; mai mare: se executa
                                  ; rutina normal
et00:
        je et0                    ; right = left + 1
        jmp gata                  ; left = right: iesire
et0:
;
; Sunt 2 elemente
; Se compara si eventual se schimba
;
        shl   si, 1               ; intregii sunt pe
        shl   di, 1               ; doi octeti
        mov   ax, [bx][si]        ; v[left]
        cmp   ax, [bx][di]        ; v[right]
        jnle  aici
        jmp   gata                ; Sunt O.K.
aici:
        xchg  ax, [bx][di]        ; Daca nu, se
        mov   [bx][si], ax        ; schimba
        jmp   gata                ; si gata
et1:
;
; Sunt mai mult de doua elemente
;
        mov   ax, si              ; Calcul
        add   ax, di              ; (left + right)/2
        shr   ax, 1
        push  bx
        shl   ax, 1               ; Adresa se aduna
        add   bx, ax              ; la v
        mov   dx, [bx]            ; mark = v[(left+right)/2]
        pop bx
_do:                              ; Ciclu do
 while_i:                         ; Ciclu while dupa
                                  ; v[i] < mark
        shl   si, 1               ; Compara v[i] cu
        cmp   [bx][si], dx        ; mark
        jge   end_i               ; Sfarsit ciclu daca
```

```
                                        ; v[i] >= mark
            shr   si, 1
            inc   si                    ; Daca nu, se face i++
            jmp   while_i               ; si se reia
            end_i:
            shr   si, 1
            while_j:                    ; Ciclu while dupa
                                        ; v[j] > mark
            shl   di, 1
            cmp   [bx][di], dx          ; Compara v[j] cu mark
            jle   end_j                 ; Sfarsit ciclu daca
                                        ; v[j] <= mark
            shr   di, 1
            dec   di                    ; Daca nu, se face j--
            jmp   while_j               ; si se reia
            end_j:
            shr   di, 1
;
            cmp   si, di                ; Compara i cu j
            jg    _while                ; Salt daca i > j
;
            shl   di, 1                 ; intregii sunt
            shl   si, 1                 ; pe doi octeti
            mov   ax, [bx][si]          ; Schimba v[i]
            xchg  ax, [bx][di]          ; cu
            mov   [bx][si], ax          ; v[j]
            shr   si, 1
            shr   di, 1
            inc   si                    ; i++
            dec   di                    ; j--
_while:
            cmp   si, di                ; Compara i cu j
            jle _do                     ; Daca i <= j se continua
                                        ; ciclul _do

            cmp   [bp].left, di         ; Compara left cu j
            jge   not_rec               ; Salt daca j <= left
                                        ; j > left: se apeleaza
                                        ; _qasm pentru
                                        ; prima partitie
;
;     qasm(v, left, j)
;
            push  di                    ; j
            push  [bp].left             ; left
            push  bx                    ; v
            call _qasm
            add   sp, 6
not_rec:
            cmp   si, [bp].right        ; Compara i cu right
            jge   gata                  ; Salt daca i >= right
                                        ; i < right: se apeleaza
                                        ; _qasm pentru
                                        ; a doua partitie
;
```

```
; qasm(v, i, right)
;
            push   [bp].right          ; right
            push   si                  ; i
            push   bx                  ; v
            call   _qasm
            add    sp,6
gata:
            popa
            pop    bp
            ret
            _qasm endp
end
;
;
; Interfata cu limbajul C se realizeaza prin functia Quick_a,
; care are un prototip asemanator cu functia Quick_C.
;
; void Quick_a(void *v, size_t n, size_t size, PFCMP cmp);
; {
;           qasm(v, 0, (int) n-1);
; }
;
```

7.7.3 Compararea performanțelor

Evaluarea performanțelor unui algoritm de sortare internă constă în estimarea sau în măsurarea numărului de operații de comparație între elemente și a numărului de operații de copiere sau interschimbare a două elemente. Aceste evaluări se efectuează în trei situații distincte de tablouri: tablou aleator, tablou deja sortat (cazul cel mai favorabil) și tablou sortat invers (cazul cel mai defavorabil).

În cazul de față, au fost considerate două dimensiuni de tablouri, de 1000 și respectiv 10000 de întregi, măsurându-se și timpul de execuție.

Pentru algoritmul de sortare rapidă considerat, s-au testat 3 variante: prima (Quick_C) este implementarea polimorfică prezentată în 7.7.1, a doua (Quick_a) este implementarea în ASM din 7.7.3, iar a treia variantă (qsort) este funcția polimorfică din biblioteca standard C. Rezultatele sunt date în tabelele 7.2, 7.3 și 7.4.

	Dimensiune = 1000				Dimensiune = 10000			
Alg.	Timp	Comp	Move	Copy	Timp	Comp	Move	Copy
Quick_C	0.3	12901	2443	608	4.3	187234	31508	6219
Quick_a	0.0	12901	2443	0	0.8	187234	31508	0
qsort	0.2	13321	-	-	2.9	172655	-	-

Tabelul 7.2 Performanțe în cazul unui tablou aleator

	Dimensiune = 1000				Dimensiune = 10000			
Alg.	Timp	Comp	Move	Copy	Timp	Comp	Move	Copy
Quick_C	0.1	8963	488	488	2.3	121821	4095	4095
Quick_a	0.0	8963	488	0	0.4	121821	4095	0
qsort	0.1	10819	-	-	2.2	145262	-	-

Tabelul 7.3: Performanțe în cazul unui tablou sortat

	Dimensiune = 1000				Dimensiune = 10000			
Alg.	Timp	Comp	Move	Copy	Timp	Comp	Move	Copy
Quick_C	0.2	8972	988	489	2.4	121832	9094	4095
Quick_a	0.0	8972	988	0	0.4	121832	9094	0
qsort	0.1	10818	-	-	2.3	145260	-	-

Tabelul 7.4: Performanțe în cazul unui tablou sortat invers

Din tabelele de mai sus rezultă că eficiența tuturor celor trei implementări este ridicată. Atunci când timpul de rulare este însă esențial, trebuie folosită implementarea specializată în limbaj de asamblare. Implementarea Quick_a asigură un timp de rulare de 3.5 ori mai scurt în cazul unui tablou aleator și de 5.5 ori mai scurt în cazul unui tablou sortat sau sortat invers. Toate implementările au utilizat modelul de memorie small.

Capitolul 8

Interfața cu sistemul de operare DOS

În acest capitol, vor fi prezentate elementele de bază ale interacțiunii programelor scrise în ASM cu cele două componente software de bază dintr-un calculator personal, subsistemul BIOS și sistemul de operare DOS, fără a se prezenta o listă exhaustivă a funcțiilor de sistem sau BIOS. Asemenea liste de funcții sunt disponibile în manuale specifice ale sistemului de operare DOS.

8.1 Componentele de bază ale sistemului DOS

Componentele sistemului de operare DOS sunt:

- **DOS-BIOS.** Această componentă este memorată pe disc într-un fișier cu numele IBMBIO.COM sau IBMIO.SYS sau IO.SYS, depinzând de furnizor. Ea conține driverele pentru dispozitivele CON (tastatură și display), PRN (imprimantă), AUX (interfață serială) și CLOCK (ceas de timp real). De asemenea, modulul DOS-BIOS conține drivere pentru discurile flexibile și pentru discul hard. Accesul la aceste drivere va implica apelul unor rutine din ROM-BIOS (porțiunea de BIOS aflată în ROM, care conține procedurile dependente de hardware).
- **DOS-Kernel (nucleul DOS)**. Această componentă este memorată pe disc în fișierul IBMDOS.COM sau MSDOS.SYS, depinzând de furnizor și conține funcții de acces la fișiere, funcții de intrare/ieșire la nivel de caracter etc. Aceste funcții operează într-un mod independent de hardware (e vorba de funcțiile DOS apelate prin instrucțiuni INT 21H).
- **Interpretorul de comenzi (Shell) standard**. Este memorat pe disc în fișierul COMMAND.COM. Interpretorul este cel care afișează prompterul DOS la consolă, acceptând și executând comenzi de la tastatură. Interpretorul de comenzi este la rândul său compus din 3 părți:
  - **porțiunea rezidentă**, care conține handler-e (rutine de tratare) pentru acțiunile CTRL-Break (CTRL-C) de la tastatură și pentru erori critice (erori la citirea sau scrierea la dispozitive periferice, cum ar fi discurile sau imprimanta). Erorile critice conduc la afișarea la consolă a unor mesaje de forma:

```
Error on device ... Abort, Retry, Fail ?
```

  - **porțiunea tranzitorie**, care afișează prompter-ul la consolă (de exemplu C:\>), citește comanda de la consolă și o execută. Porțiunea

tranzitorie conține comenzile DOS interne (de genul DIR, TYPE, COPY etc.). Când un program executabil își încheie execuția, controlul revine porțiunii rezidente a interpretorului de comenzi, care verifică o sumă de control, detectând astfel dacă s-a modificat cumva porțiunea tranzitorie din memorie. În caz afirmativ, porțiunea tranzitorie se reîncarcă de pe disc;

- **rutina de inițializare**, care este încărcată în memorie o dată cu încărcarea sistemului DOS de pe disc. Când operația de încărcare s-a terminat, această componentă nu mai este necesară și spațiul respectiv de memorie va fi folosit în alte scopuri.

Comenzile acceptate de interpretorul de comenzi standard sunt de 3 tipuri:

- comenzi interne (cum ar fi COPY, DIR, RENAME etc.), care nu au corespondent în fișiere executabile pe disc;
- comenzi externe, conținute în fișiere disc de tip .EXE sau .COM, care sunt încărcate în memorie în zona TPA (zona de procese tranzitorii);
- fișiere de comenzi indirecte (.BAT), care sunt fișiere text ce conțin comenzi interne, externe sau alte invocări de fișiere de comenzi indirecte.

Interpretorul testează întâi dacă linia introdusă de la tastatură este o comandă internă. Dacă nu este, se caută un fișier cu extensia .COM, .EXE sau .BAT (în această ordine), întâi în directorul curent, apoi în toate directoarele specificate în comanda PATH. Dacă nu se găsește nici un fișier adecvat, se afișează la consolă mesajul:

```
Bad command or file name
```

Dacă s-a identificat un fișier cu extensia .COM sau .EXE, interpretorul încarcă fișierul executabil în memorie utilizând funcția EXEC (4BH) a sistemului de operare și îi dă controlul. Funcția EXEC construiește în memorie un bloc special numit PSP (Program Segment Prefix), în zona de procese tranzitorii TPA, la prima adresă disponibiă. Se încarcă apoi programul executabil, imediat după PSP și se execută operațiile de relocare, adică se ajustează valorile simbolurilor care conțin referințe la memorie, conform cu poziția fizică din memorie unde a fost încărcat programul.

- **ROM-BIOS**. Această componentă este rezidentă în memoria ROM a calculatorului și conține partea de autotestare de la punerea sub tensiune (numită și POST, de la **Power-On Self Test**), programul primar de încărcare de pe disc (**bootstrap**), rutinele de intrare/ieșire de nivel scăzut (dependente de hardware) etc. În general, această componentă nu rezidă numai pe placa de bază a calculatorului, anumite porțiuni fiind localizate pe plachetele de interfețe furnizate de producător.

8.2 Încărcarea sistemului DOS

La punerea sub tensiune, procesorul începe să execute instrucțiuni de la adresa fixă 0FFFF0H (care este situată în memoria de tip ROM). La această adresă se găsește o instrucțiune JMP FAR la o porțiune de program din ROM-BIOS, numită **POST** (**Power- On Self Test**). Acest cod execută teste hardware pentru unitatea centrală, memoria internă, dispozitivele periferice etc., înscriind totodată vectorii de întrerupere pentru întreruperile hardware externe.

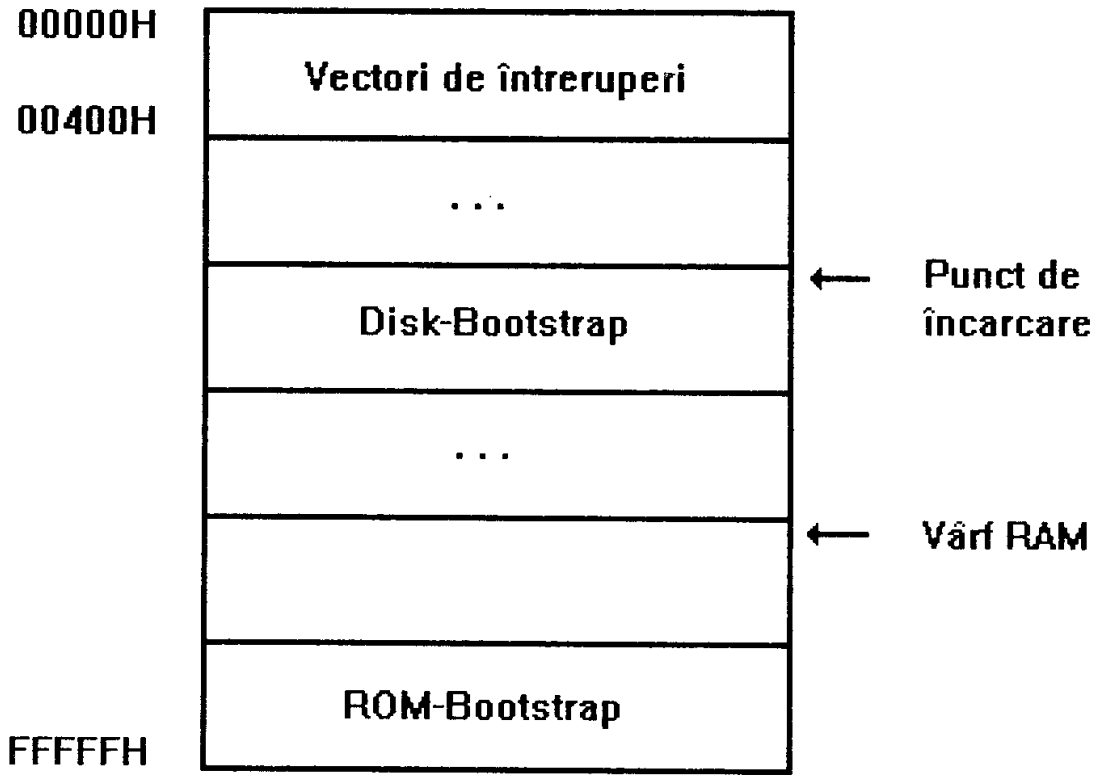


Figura 8.1 Harta memoriei imediat după încărcarea programului Disk-Bootstrap

Se trece apoi la execuția unei zone de cod numită **ROM-Bootstrap** (încărcător primar), care testează dacă există dischete în unitățile de disc flexibil. Dacă nu, se trece la examinarea discului hard. Se citește apoi primul sector de pe pista 0 a discului (fie de pe discul flexibil, fie de pe discul hard). Acest sector este numit sector de boot (**Boot Sector**) și conține un alt program încărcător (numit **Disk-Bootstrap**), precum și informații despre organizarea discului. Programul **Disk-Bootstrap** este citit în memorie la nivel fizic și i se dă controlul. Unele calculatoare de generație mai nouă permit specificarea ordinii de examinare a unităților de disc.

Trebuie remarcat că, în acest moment, sistemul de operare nu este încărcat în memorie, deci nu se dispune de nici o funcție DOS. Harta memoriei la acest moment este ilustrată în Figura 8.1.

Programul încărcător citește primul sector al directorului rădăcină, determinând dacă primele două fișiere din acest director sunt, în ordine: **IO.SYS** (sau **IBMBIO.COM** sau **IBMIO.SYS**) și **MSDOS.SYS** (sau **IBMDOS.COM**), care conțin componentele **DOS-BIOS** și **DOS-Kernel** ale sistemului de operare. Dacă aceste fișiere nu sunt identificate, se afișează la consolă mesajul:

```
Non-System disk or disk error
Replace and strike any key when ready
```

Dacă fișierele sunt identificate corect, programul **Disk-Bootstrap** le citește în memorie și transferă controlul la punctul de start al modulului **IO.SYS**.

Componenta **DOS-BIOS** constă din două module: **BIOS**-ul propriu-zis și modulul **SYSINIT**. Harta memoriei din acest moment este ilustrată în Figura 8.2.

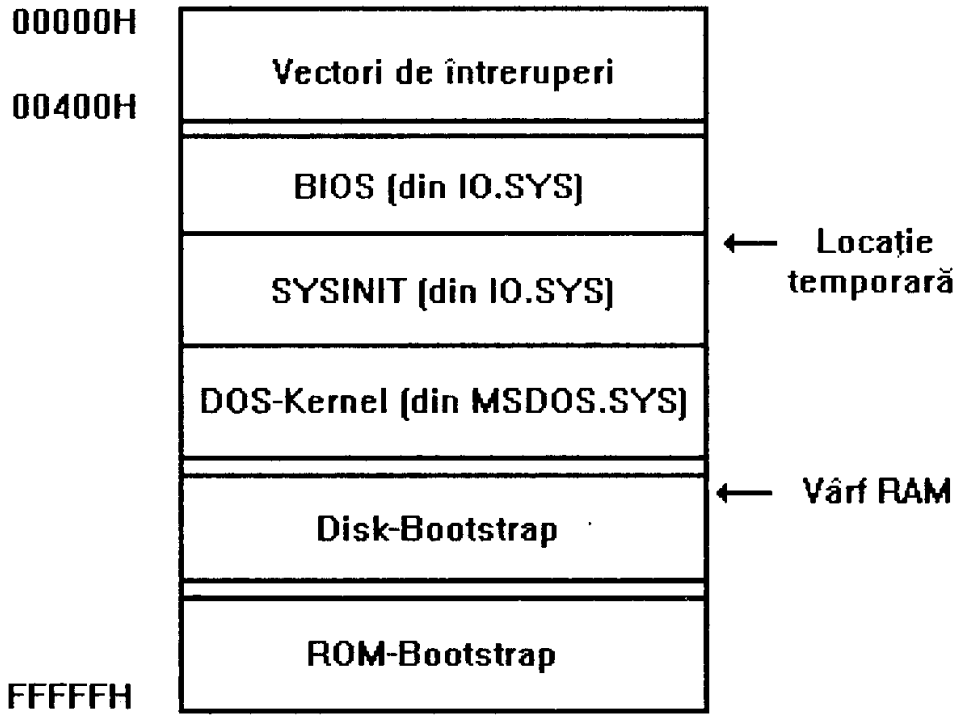


Figura 8.2 Harta memoriei după încărcarea modulelor IO.SYS și MSDOS.SYS

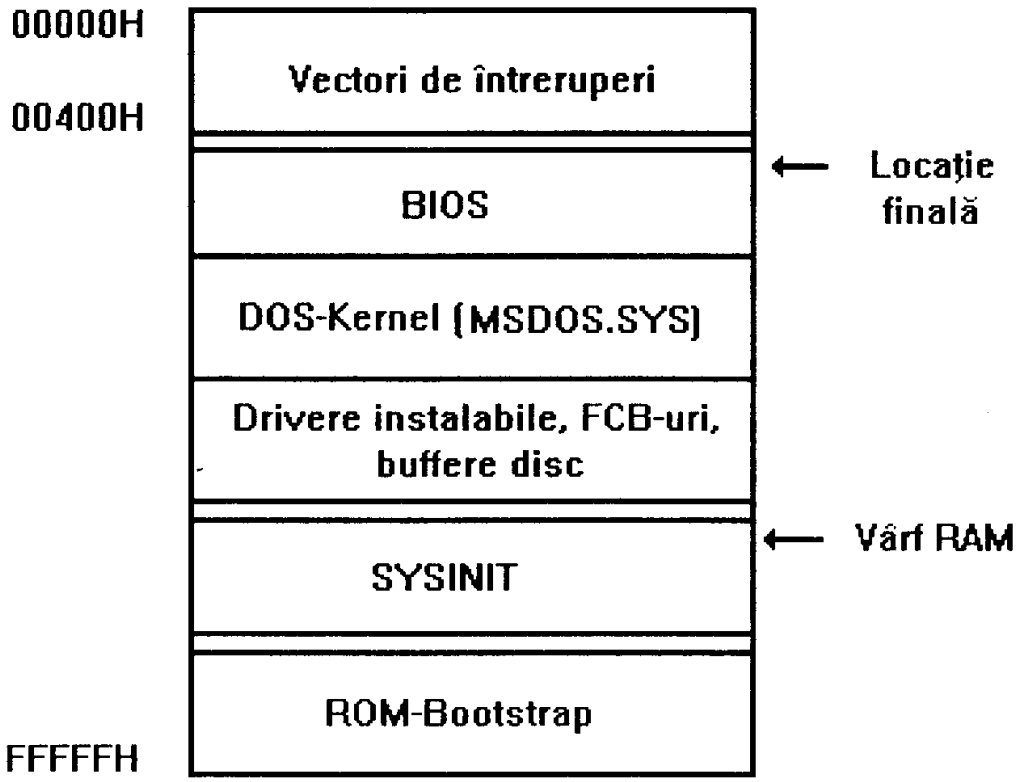


Figura 8.3 Harta memoriei înainte de încărcarea interpretorului de comenzi

Încărcarea celor două fișiere se face la adrese mici de memorie, deoarece nu se cunoaște încă dimensiunea memoriei. Modulul **SYSINIT**, apelat de către partea de inițializare a **BIOS**-ului, determină dimensiunea memoriei continue și se auto-realocă (adică se mută) la o adresă mare de memorie (către coada memoriei). În continuare, modulul **SYSINIT** mută modulul **MSDOS.SYS** din locul unde a fost încărcat inițial în poziția sa finală din memorie, suprascriind vechea copie a modulului **SYSINIT** (încărcat inițial la adrese mici), precum și alte porțiuni inițiale de cod conținute în **IO.SYS** și care nu mai sunt necesare.

Se dă apoi controlul porțiunii de inițializare a modulului **DOS-Kernel**. Acesta inițializează tabelele interne și zonele de lucru ale sistemului **DOS**, setează vectorii de întrerupere de pe nivelele 20H...2FH (adică întreruperile soft specifice funcțiilor **DOS**) și inițializează driverele rezidente ale dispozitivelor periferice (discuri, consolă etc.). Controlul revine apoi modulului **SYSINIT**.

În acest moment, sistemul DOS este operațional, deci modulul **SYSINIT** poate apela funcții **DOS**. **SYSINIT** încearcă să deschidă fișierul **CONFIG.SYS**, care conține diverse comenzi specifice prin care se poate configura sistemul. În acest fișier, se pot specifica drivere suplimentare care să fie încărcate (așa-numitele drivere instalabile), se poate specifica numărul de buffere pentru disc, numărul maxim de fișiere care pot fi deschise simultan, numele fișierului care conține

interpretorul de comenzi etc. Dacă fișierul **CONFIG.SYS** este identificat în directorul rădăcină, el este încărcat în memorie și se execută fiecare comandă specificată. Harta memoriei din acest moment este ilustrată în Figura 8.3.

După ce toate driverele instalabile au fost încărcate, **SYSINIT** închide toate fișierele (dispozitivele) folosite și apoi redeschide dispozitivele consolă (CON), imprimantă (PRN) și interfață serială (AUX), asociindu-le handler-ele de fișiere 0 (standard input), 1 (standard output), 2 (standard error), 3 (standard list) si 4 (standard auxiliary). Handler-ele 0, 1, 2 sunt asignate la dispozitivul CON, handler-ul 3 este asignat la dispozitivul PRN, iar handler-ul 4 la dispozitivul AUX. Aceste handler-e pot fi reasignate prin apeluri de funcții sistem adecvate.

În final, modulul **SYSINIT** apelează funcția **EXEC** a sistemului de operare, pentru a încărca și a executa interpretorul standard de comenzi. O dată ce interpretorul a fost încărcat, el afișează la consolă prompterul specific și așteaptă introducerea de comenzi de la tastatură. Porțiunea tranzitorie din **COMMAND.COM** este încărcată peste **SYSINIT**, care nu mai este necesar. În final, imaginea memoriei este cea din Figura 8.4.

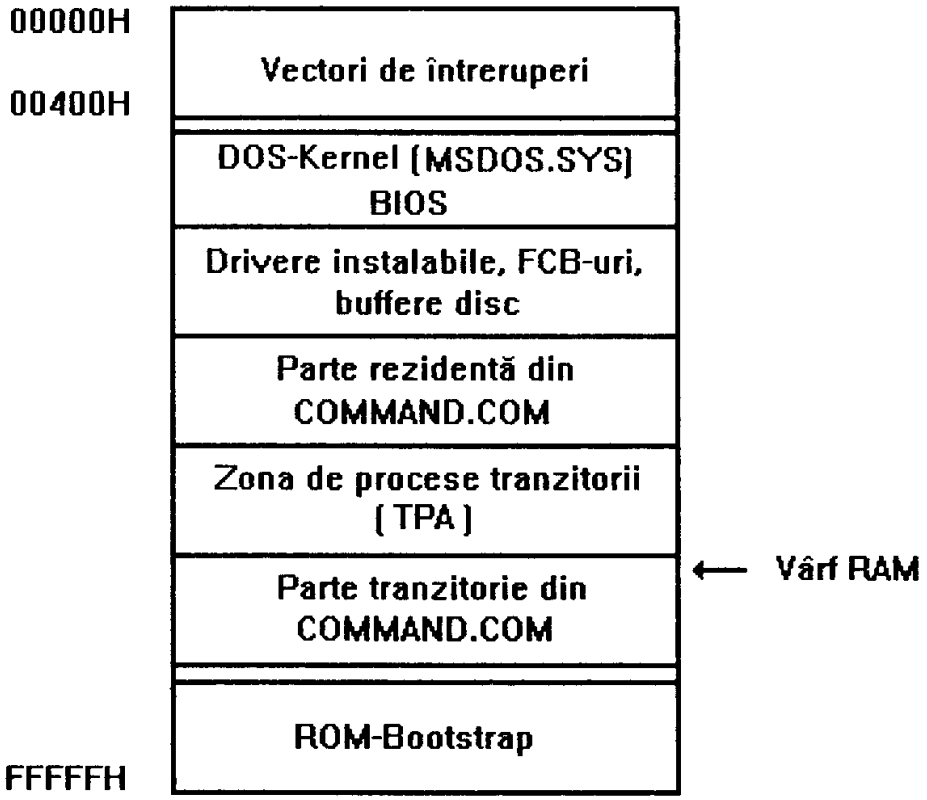


Figura 8.4 Imaginea finală a memoriei

8.3 Structura blocului PSP (Program Segment Prefix)

Blocul **PSP** este caracteristic ambelor categorii de programe executabile (**.EXE** și **.COM**), fiind o zonă de 256 de octeți care precede imaginea din memorie a programului executabil. La lansarea în execuție a unui program, adresele DS:0000 și ES:0000 indică începutul blocului **PSP** asociat programului. PSP-ul este construit de către funcția **EXEC** la încărcarea programului de pe disc. Structura **PSP**-ului este ilustrată în Figura 8.5.

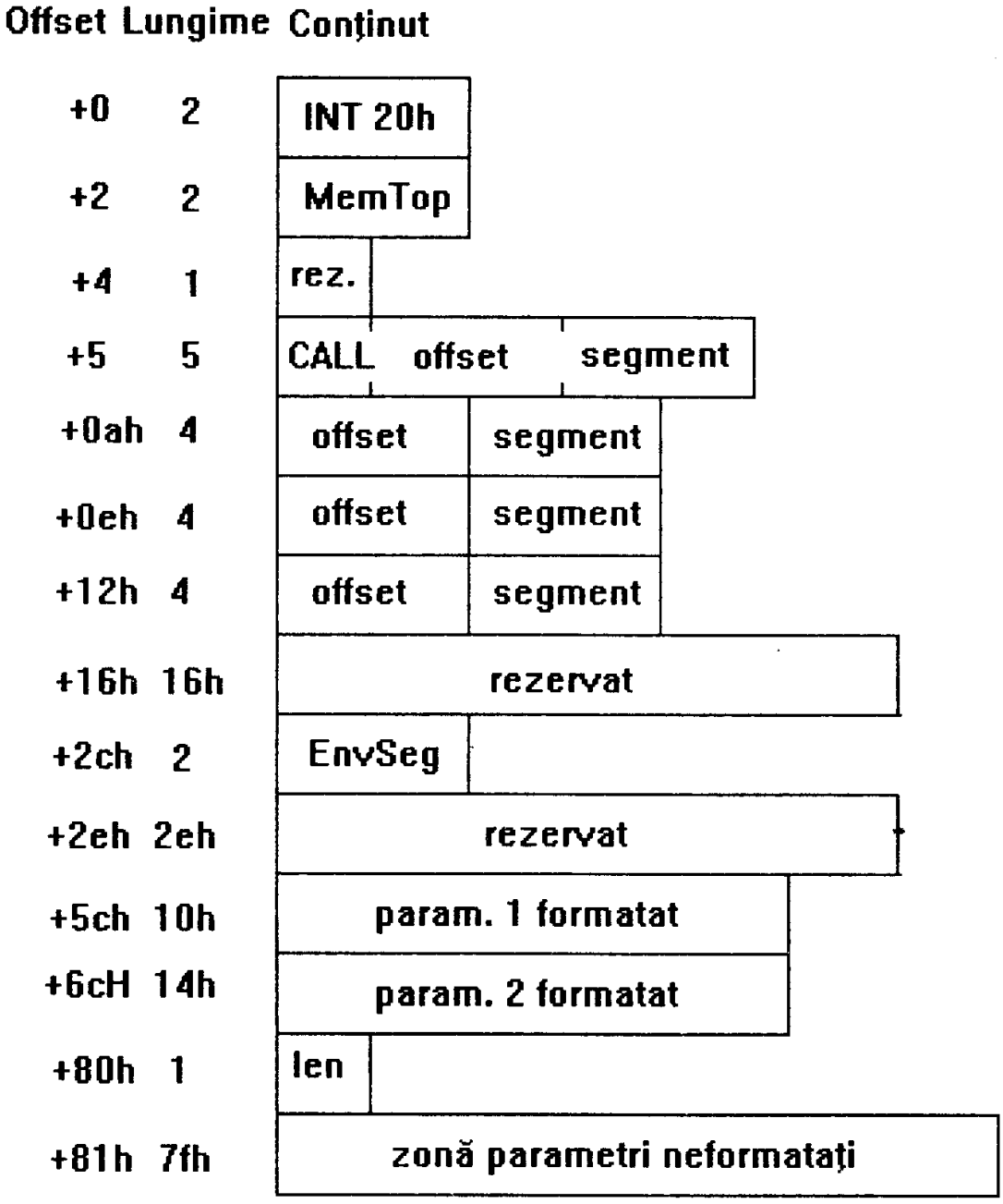


Figura 8.5 Structura blocului PSP

Semnificația câmpurilor din Figura 8.5 este următoarea:

- *octeții 0-1* - conțin codul unei instrucțiuni INT 20H (ieșire în DOS), care permite unui program executabil să se încheie cu o instrucțiune JMP sau RET la adresa PSP:0; această tehnică nu mai este actuală, pentru terminarea unui program utilizându-se funcția DOS 4CH;
- *octeții 2-3* - conțin dimensiunea memoriei de bază continue (vârful memoriei), în paragrafe de câte 16 octeți; această informație este folosită în procesul de încărcare a programului executabil;
- *octetul 4* - rezervat;
- *octeții 5-9* - conțin codul unei instrucțiuni CALL de tip far la dispecerul de funcții DOS;
- *octeții 0AH-0DH* - conțin adresa far a rutinei de încheiere (accesibilă prin INT 22H); întreruperea 22H este o altă modalitate perimată de a încheia programele executabile; actualmente se utilizează funcția DOS 4CH;
- *octeții 0EH-10H* - conțin adresa far a rutinei de tratare a apăsării tastelor Ctrl-Break (accesibilă prin INT 23H); Ctrl-Break provoacă terminarea forțată a programelor în curs de execuție;
- *octeții 12H-15H* - conțin adresa far a rutinei de tratare a erorilor critice (accesibilă prin INT 23H); (erorile critice sunt de genul: nu există dischetă în unitatea de disc flexibil și se dă o comandă de genul DIR A:);
- *octeții 16H-2BH* - rezervați;
- *octeții 2CH-2DH* - conțin adresa de segment a blocului de environment; acesta este o zonă de memorie în format text, care conține informații de genul: căi (path) active, specificarea interpretorului de comenzi, specificarea prompterului de comenzi etc; între altele, în blocul de environment se găsește calea completă a fișierului care conține programul executabil; în felul acesta, se poate ști în program numele fișierului executabil din care a fost încărcat programul; acesta este suportul fizic al argumentului argv[0] al funcției main de la programele C;
- *octeții 2EH-5BH* - rezervați;
- *octeții 5CH-6BH și 6CH-7FH* - reprezintă așa numiții parametri formatați; sunt utilizați pentru două blocuri de control fișiere (FCB-uri), care reprezintă o modalitate perimată de lucru cu fișiere disc; actualmente se utilizează funcții de prelucrare a fișierelor orientate pe handlere (vezi 8.7);
- *octetul 80H* - reprezintă lungimea (numărul de caractere) conținute în zona de parametri neformatați;
- *octeții 81H-0FFH* - reprezintă zona de parametri neformatați; această zonă conține caracterele introduse în linia de comandă după numele fișierului executabil (așa numita coadă a comenzii); prin această zonă, un program executabil are acces la parametrii din linia de comandă (acesta este suportul oferit de DOS pentru parametrii numiți generic argc și argv de la

funcția main a programelor C); parametrii din linia de comandă nu cuprind parametrii de redirectare a dispozitivelor standard de intrare și de ieșire; primii 43 de octeți de la offset-ul 80H mai sunt utilizați implicit și **ca Zonă de Transfer Disc (DTA)**; această zonă este folosită de unele funcții de acces la disc și se poate poziționa într-un spațiu rezervat de către utilizator prin funcția DOS 1AH; de asemenea, adresa curentă a zonei DTA poate fi citită cu funcția DOS 2FH;

8.4 Structura programelor de tip COM

Programele de tip COM corespund modelului de memorie tiny, în care există un singur segment și toate salturile și apelurile de proceduri sunt de tip near. Fișierul disk conține exclusiv modulul care se încarcă, fără alte informații suplimentare, ca în cazul fișierelor de tip EXE.

Imaginea memoriei unui program COM este prezentată în Figura 8.6.

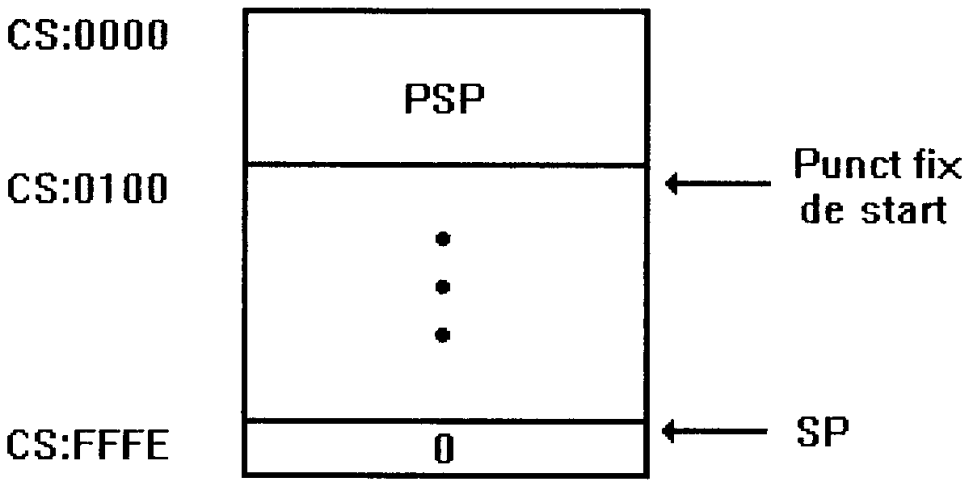


Figura 8.6 Imaginea memoriei ocupate de un program COM

Programele COM au ca punct de start adresa near 100H, adică imediat după blocul PSP. Indicatorul SP este plasat la adresa 0FFFEH, deci pe ultimul cuvânt adresabil. Acest cuvânt e inițializat cu 0.

Un program COM se poate încheia în unul din următoarele moduri:

- prin apel al funcției standard de terminare (funcția DOS 4CH)
- prin apel al întreruperii soft INT 20H
- prin execuția unei instrucțiuni RETN, dacă registrul SP conține valoarea inițială 0FFFEH. Cuvântul de la adresa 0FFFEH fiind 0, instrucțiunea RETN

va da controlul la adresa near 0, adică la primul octet din PSP. Acolo însă se găsește codul unei instrucțiuni INT 20H, care provoacă revenirea în DOS.

Programele COM nu necesită relocare, deoarece toate simbolurile relative sunt deplasamente în cadrul unui segment unic. Simpla încărcare a registrului CS este suficientă.

Șablonul de dezvoltare pentru programele COM este:

```
_TEXT               segment para 'CODE'
      ORG           100H
      ASSUME        CS:_TEXT, DS:_TEXT, ES:_TEXT, SS:_TEXT
start:
            jmp     init
            ;
            ; Aici se pun eventuale definitii de date
            ;
init:
            ;
            ; Aici se pune codul propriu-zis
            ;
            mov     ax, 4C00H                   ; Apel functie DOS
            int     21H                         ; pentru terminare
                                                ; program
_TEXT ENDS
end   start                                     ; Eticheta de start
```

Programele de tip COM (ca de altfel și cele de tip EXE) își rezervă toată memoria disponibilă. În unele situații, e necesară reducerea la minim a memoriei rezervate, care se poate realiza printr-un apel al funcției DOS 4AH. Șablonul de dezvoltare al programelor COM va fi următorul:

```
_TEXT               segment para 'CODE'
      ORG           100H
      ASSUME        CS:_TEXT, DS:_TEXT, ES:_TEXT, SS:_TEXT
start:
            jmp     init
            ;
            ; Aici se pun definitii de date
            ;
main  proc near
            ;
            ; Aici se pune codul propriu-zis
            ;
            mov     ax, 4C00H
            int     21H
            main    endp
            ;
            ; Aici se pun alte proceduri (daca e cazul)
            ;
init:
            mov     ah, 4AH                     ; Functie redimensionare
                                                ; memorie
```

```
                mov     bx, offset end_mem      ; Limita pana unde
                                                ; se pastreaza
                mov     cl, 4                   ; Dimensiunea memoriei
                                                ; se specifica in
                shr     bx, cl                  ; paragrafe de
                                                ; 16 octeti
                inc     bx                      ; Mai punem unul
                                                ; de siguranta
                int     21H                     ; Apel propriu-zis
                mov     sp, offset end_mem      ; Initializare sp
                jmp     main
;
; Rezervam o stiva de 512 octeti, care sa cuprinda
; si spatiul dintre etichetele init si init_mem, care se poate
; suprascrie dupa incarcarea programului
init_mem:
                db (512 - (init_mem - init)) dup (?)
;
; Pana aici rezervam memoria
;
end_mem:
_TEXT ENDS
end     start
```

8.5 Structura și încărcarea fișierelor EXE

Programele de tip EXE pot fi încărcate în orice locație de memorie și se compun din mai multe segmente. Acest fapt impune ajustarea tuturor instrucțiunilor care conțin adrese de segment, cum ar fi:

- JMP/CALL FAR PTR nume;
- MOV reg, SEG nume;
- MOV reg_seg, valoare.

Acest proces se numește **relocare** și se execută la încărcarea programului în memorie. Punctul de start poate fi oarecare și este stabilit prin directiva END.

Este evident că un fișier de tip EXE trebuie să conțină, pe lângă programul propriu-zis, și informații despre simbolurile relocabile, adresa de start, adresa segmentului de stivă etc. Aceste informații sunt grupate în prima parte a fișierului, numită **header al fișierului EXE**. Structura acestui header este ilustrată în Tabelul 8.7.

Offset	Dim.	Conținut		Semnificație
0	2	4DH 5AH		Semnătură fișier .EXE ('MZ')
2	2	PartPag		Lungime fișier modulo 512
4	2	PageCnt		Lungime în pagini de 512 octeți, inclusiv header-ul
6	2	ReloCnt		Număr de elemente din tabela de relocare
8	2	HdrSize		Dimensiune header în paragrafe de 16 octeți
0AH	2	MinMem		Necesar minim de memorie peste sfârșitul programului (în paragrafe)
0CH	2	MaxMem		Necesar maxim de memorie peste sfârșitul programului (în paragrafe)
0EH	2	ReloSS		Deplasament segment stivă
10H	2	ExeSP		Valoare SP la execuție
12H	2	ChkSum		Sumă de control
14H	2	ExeIP		Valoare IP (adresă de start)
16H	2	ReloCS		Deplasament segment cod
18H	2	TablOff		Poziția în fișier a tabelei de relocare (uzual 1CH)
1AH	2	Overlay		Indicator overlay (0 pentru modulele de bază)
?	4*	offs ... offs	seg ... seg	Tabela de relocare
?	?			Caractere până la limita de paragraf

Tabelul 8.7 Structura headerului fișierelor .EXE

Primii doi octeți identifică tipul fișierului. Următorii 4 octeți conțin lungimea fișierului (inclusiv header-ul), în pagini de 512 octeți și, respectiv, modulo 512.

Suma de control reprezintă suma cu semn schimbat a tuturor cuvintelor din fișier, permițând un control al validității fișierului.

Valorile ReloSS și ReloCS reprezintă deplasamentele segmentelor de stivă și cod față de adresa de segment de încărcare (segmentele pot fi în orice ordine). Valorile ExeSP și ExeIP reprezintă conținutul registrelor SP și IP la intrare în execuție. Aceste valori sunt deduse din dimensionarea segmentului de stivă și din directiva END a modulului de program principal (care precizează eticheta de start).

Tabela de relocare cuprinde adresele tuturor cuvintelor care trebuie ajustate, adrese precizate prin offset în cadrul segmentului curent și prin deplasamentul segmentului respectiv față de adresa de segment inițială. Tabela de relocare are ReloCnt (vezi [6]) elemente, începe la poziția TablOff (vezi [18H]) în fișier și ocupă ReloCnt•4 octeți.

Header-ul este completat cu octeți fără semnificație, până la o limită de paragraf.

Relocarea programului este realizată de funcția EXEC a sistemului de operare (funcția 4BH) și constă în următorul algoritm:

- Se creează un PSP cu funcția DOS 26H;
- Se citesc 1CH octeți din fișierul .EXE (așa-numita porțiune formatată a header-ului), într-o zonă locală de memorie;
- Se determină lungimea modulului = ((PageCnt • 512) - (HdrSize • 16)) - PartPag;
- Se determină deplasamentul în fișier al modulului = (HdrSize • 16);
- Se selectează o adresă de segment START_SEG (uzual PSP + 10H);
- Se citește modulul de program în memorie la adresa START_SEG:0000;
- Se setează poziția de citire din fișier la începutul tabelei de relocare (TablOff);
- Pentru fiecare element al tabelei de relocare (ReloCnt):
  - se citește elementul din tabelă ca două cuvinte (I_OFF, I_SEG);
  - se determină adresa actuală de memorie a elementului care trebuie relocat RELO_SEG = (START_SEG + I_SEG);
  - se citește elementul care trebuie relocat, deci cuvântul de la adresa (RELO_SEG:I_OFF);
  - se adună START_SEG la acest cuvânt (se ajustează adresa de segment);
  - se depune valoarea ajustată înapoi la adresa (RELO_SEG:I_OFF);
- Se alocă memorie suplimentară pentru program, conform MaxMem și MinMem;
- Se inițializează registrele semnificative și se dă controlul programului:
  - ES = DS = PSP;

- SS = START_SEG + ReloSS;
- SP = ExeSP;
- CS = START_SEG + ReloCS;
- IP = ExeIP.

Poziționarea registrelor CS și IP se realizează printr-o secvență de forma:

```
PUSH  START_SEG + ReloCS
PUSH  ExeIP
RETF                              ; Intrare in executie
```

8.6 Programe de tip TSR (Terminate and Stay Resident)

Programele de tip EXE sau COM obișnuite sunt încărcate în memorie în zona de procese tranzitorii (TPA) și ocupă memoria până la încheierea execuției. Mai concret spus, la execuția funcției DOS 4CH, memoria ocupată de program este eliberată și oferită sistemului DOS.

Programele de tip TSR (care sunt tot programe de tip EXE sau COM) rămân în memorie o durată nedeterminată de timp (sunt **rezidente**). Ele sunt **activate** de către evenimente externe, cum ar fi acționarea unei taste special destinate (**hot-key**) sau expirarea unui interval de timp. Pentru ca să rămână rezident, un program trebuie să-și încheie execuția cu un apel al funcției DOS 31H, în care se precizează (în registrul DX) numărul de paragrafe de memorie care se păstrează rezidente.

Se deosebesc, deci, următoarele noțiuni:

- **activarea unui program TSR** - reprezintă întreruperea programului care se rulează în mod curent (nu neapărat un program utilizator), cu salvarea corespunzătoare a întregului context și transferul controlului către programul TSR;
- **dezactivarea unui program TSR** - reprezintă refacerea contextului programului întrerupt și transferul controlului către acesta;
- **instalarea unui program TSR** - reprezintă încărcarea de pe disc și încheierea execuției prin apelul funcției 31H;
- **dezinstalarea unui program TSR** - reprezintă eliberarea memoriei ocupate de un program TSR instalat anterior.

8.6.1 Activarea și dezactivarea programelor TSR

Activarea se face, de obicei, ca urmare a acționării unei taste special destinate. Se utilizează întreruperea pe nivelul 9 (asociat tastaturii), care se produce la fiecare apăsare și ridicare a unei taste.

Interacțiunea cu întreruperea 9 se face prin modificarea („furtul") vectorului de întrerupere cu o rutină proprie de tratare, care apelează rutina veche, testând dacă s-a apăsat tasta specială de activare a TSR-ului. Rutina veche trebuie

apelată, deoarece ea convertește codul de scanare produs de tastatură într-un cod ASCII, actualizează buffer-ul tastaturii, gestionează starea tastelor Ctrl, Shift, Alt etc. De asemenea, este posibil să fie instalate mai multe TSR-uri „unul peste altul", activate prin taste speciale diferite, care au modificat la rândul lor vectorul 9 de întrerupere. Prin apelul rutinei vechi, se iau în considerare și aceste TSR-uri instalate anterior.

Apelul rutinei vechi se face simulând o instrucțiune INT, prin secvența:

```
pushf
call  dword ptr cs:oldint
```

unde oldint este o locație în care s-a salvat adresa vechii rutine de tratare.

Trebuie observat faptul esențial că, la activarea unui TSR, dintre registrele de segment nu se poate conta decât pe conținutul registrului CS. Acesta este poziționat în urma invocării rutinei de tratare pe nivelul 9, în timp ce restul registrelor de segment sunt poziționate conform programului care a fost întrerupt. De aceea, programele TSR se dezvoltă de obicei sub forma unui unic segment, adresabil prin registrul CS. Aceasta explică prezența prefixului de segment din instrucțiunea CALL de mai sus. Dacă este cazul, se pot încărca și alte registre de segment cu valori adecvate, dar aceste valori trebuie memorate la etapa de instalare, obligatoriu în locații din segmentul adresabil prin registrul CS.

Secvența de parcurgere a mai multor programe TSR instalate este ilustrată în Figura 8.8.

Non-reentranța funcțiilor DOS

Problemele speciale legate de activarea TSR-urilor sunt în principal datorate faptului că sistemul DOS **nu este reentrant**. Aceasta înseamnă că, dacă se întrerupe execuția unei funcții DOS și din rutina de întrerupere se apelează o altă funcție DOS, este posibil ca funcționarea sistemului de operare să fie compromisă.

Soluția constă în activarea TSR-urilor numai în situații „sigure", adică, teoretic, atunci când nu se execută funcții DOS. Se utilizează o combinație de două tehnici:

- Prima tehnică se bazează pe un flag de sistem, numit uzual flag INDOS, care menține un contor al nivelurilor de adâncime ale apelurilor de funcții DOS. Adresa flagului INDOS poate fi obținută prin funcția DOS 34H. Teoretic, ar trebui să activăm TSR-ul doar atunci când acest contor este nul. Din păcate, flagul INDOS este diferit de zero mai tot timpul, datorită interpretorului de comenzi care așteaptă introducerea comenzilor de la tastatură prin execuția funcției DOS 0AH (citire șir de caractere).

- A doua tehnică se bazează pe întreruperea soft 28H, care este apelată periodic de către sistem atunci când interpretorul așteaptă comenzi de la consolă. Se consideră că, în momentul apelului acestei întreruperi, este relativ sigur să se activeze programul TSR.

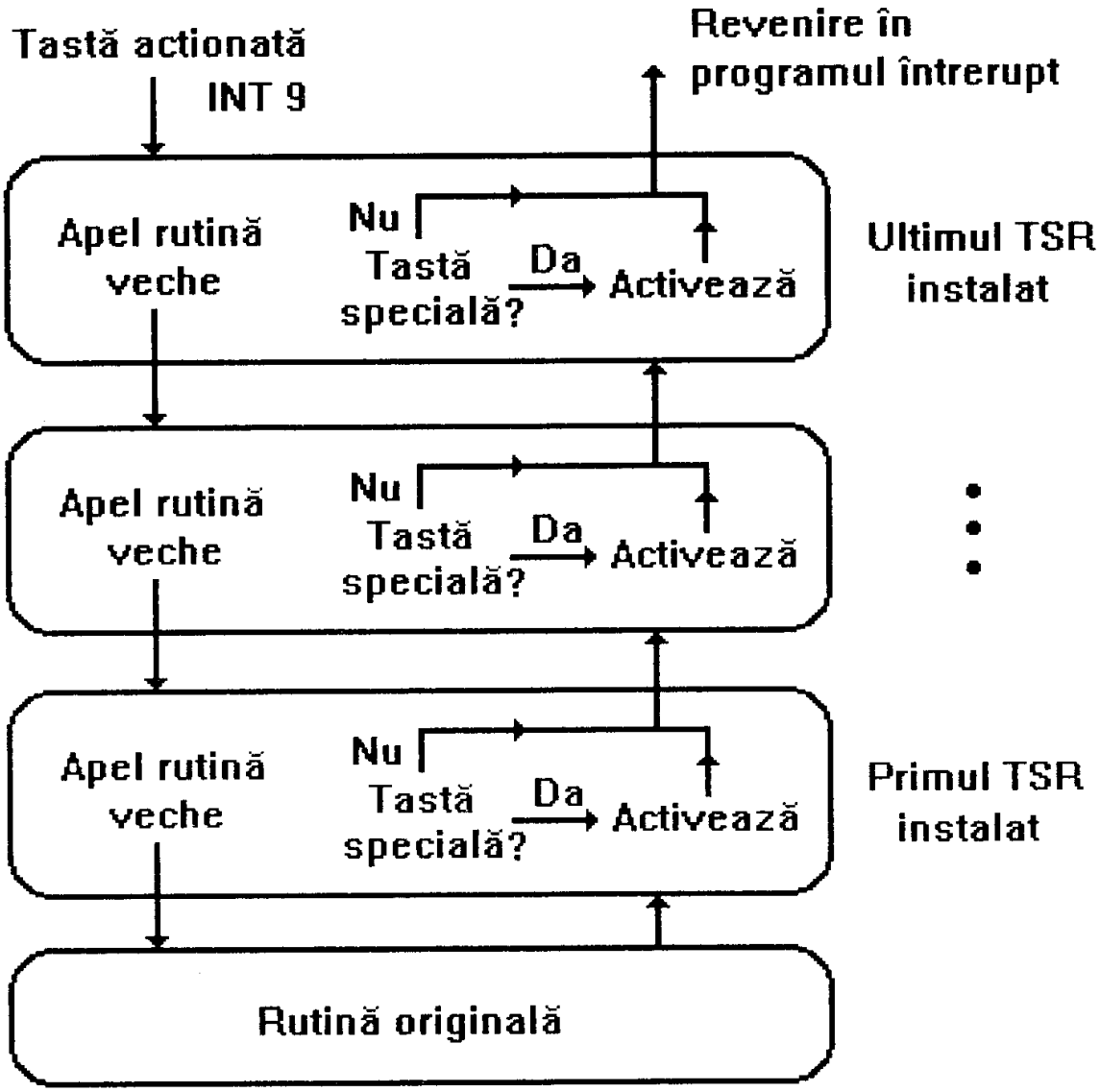


Figura 8.8 Parcurgerea succesivă a mai multor rutine de tratare pe nivelul 9

În loc de a activa programul TSR imediat după ce a apărut cererea de activare, se va memora această cerere într-o variabilă, urmând ca ulterior, când sunt îndeplinite condițiile de activare, să se facă activarea propriu-zisă. Concret, se va modifica rutina de tratare pe nivelul 28H, astfel încât să se testeze variabila care memorează cererea de activare, și, dacă există cerere, să se activeze TSR-ul.

Acțiuni critice în timp

O altă categorie de probleme o reprezintă **acțiunile critice în timp**. Acestea sunt activități ale sistemului de operare sau ale BIOS-ului, a căror întrerupere ar putea duce la funcționări defectuoase. Concret, este vorba despre rutinele de nivel scăzut pentru operațiile cu discurile, realizate prin întreruperea de BIOS 13H și de operațiile de nivel scăzut asupra display-ului, realizate prin întreruperea BIOS 10H.

Problema se rezolvă prin modificarea întreruperilor 13H și 10H, adăugându-se poziționarea la 1 a unui flag propriu, pe durata cât aceste rutine de tratare sunt active. Programul TSR trebuie activat doar dacă flagul respectiv este nul. Structura noilor rutine de tratare va fi:

```
new13 proc  far
      mov   flag_bios, 1        ; Marcheaza operatii
                                ; critice in curs
      pushf                     ; Simulare executie
      call  dword ptr cs:old13; apel INT 13H
      mov   flag_bios, 0        ; Marcheaza sfarsit
                                ; operatii critice
      retf  2                   ; Revenire cu flagurile
                                ; intoarse de
                                ; apelul lui old13
new13 endp
```

în care old13 este o variabilă DWORD care memorează adresa vechii rutine de tratare, iar flag_bios este o variabilă rezervată în segmentul adresabil prin CS. Se observă încheierea procedurii prin RETF 2, și nu prin IRET (deși este procedură de tratare a unei întreruperi), ceea ce menține flagurile așa cum au fost întoarse de apelul vechii rutine de pe nivelul 13H.

Evitarea activării recursive

O problemă de bază la implementarea programelor TSR o reprezintă **evitarea activării recursive**. Dacă programul TSR a fost activat și se află în execuție, este posibil să apară o nouă cerere de activare (de exemplu, se apasă din nou tasta specială de activare). În această situație, programul TSR nu trebuie reactivat. Problema se rezolvă prin menținerea unui alt flag propriu, care este 1 pe durata cât TSR-ul este activ.

Comutarea contextului la activare și la dezactivare

Programele TSR pot fi destul de complexe, în sensul că pot apela orice funcție DOS, pot lucra cu fișiere disc etc. Aceasta presupune ca activarea să fie însoțită de o salvare cât mai completă a contextului programului întrerupt și de o comutare a contextului către programul TSR. Similar, la dezactivarea TSR-ului, trebuie refăcut complet contextul programului întrerupt.

Problemele care trebuie avute în vedere sunt următoarele:

- salvarea și restaurarea registrelor programului întrerupt;
- salvarea și restaurarea unor structuri de date specifice programului întrerupt și comutarea pe structurile specifice programului TSR:
  - ♦ salvarea adresei PSP-ului programului întrerupt și marcarea ca fiind activ a PSP-ului programului TSR;
  - ♦ salvarea zonei DTA și comutarea pe zona DTA a TSR-ului;
- comutarea pe stiva proprie a programului TSR;
- salvarea unei porțiuni (uzual 64 de octeți) din stiva programului întrerupt într-o zonă proprie;
- salvarea conținutului ecranului.

Toate operațiile de mai sus trebuie realizate în sens invers la cedarea controlului către programul întrerupt.

Pentru citirea și marcarea PSP-ului și a zonei DTA sunt disponibile funcții DOS specifice. Salvarea unei porțiuni din stiva programului întrerupt este necesară, deoarece sistemul DOS lucrează cu stive proprii alocate la adrese fixe și este posibil ca întreruperea să fi apărut atunci când registrele SS:SP erau poziționate pe o asemenea stivă. Un apel al unei alte funcții DOS va utiliza poate aceeași stivă DOS, ceea ce va distruge conținutul stivei apelului anterior.

Salvarea și restaurarea ecranului apare ca necesară în situația în care TSR-ul afișează ceva la consolă (ceea ce se întâmplă destul de frecvent). Salvarea presupune un transfer între memoria video și o zonă proprie. De obicei, programele TSR lucrează în mod text, deci se va utiliza memoria video de la adresa 0B0000H (la ecrane monocrom) și 0B8000H la ecrane color. Funcție de modul video curent, se salvează 25 sau 43 de linii de ecran, fiecare a 80 de cuvinte (un simbol pe ecran este codificat prin caracterul ASCII extins și printr-un octet de atribut, care precizează culoarea sau efecte speciale de afișare). Este indicat să se salveze și modul video curent, precum și poziția cursorului pe ecran. Dacă este instalat un driver de mouse, trebuie salvată poziția cursorului mouse-ului și inhibată afișarea mouse-ului pe durata cât TSR-ul este activ. Dacă este cazul, se poate comuta pe o rutină (handler) proprie de tratare a evenimentelor de la mouse.

Pentru citirea și setarea cursoarelor, a modurilor video etc., se utilizează funcții BIOS specifice.

8.6.2 Instalarea și dezinstalarea programelor TSR

Instalarea unui program TSR înseamnă încărcarea lui în memorie și pregătirea condițiilor pentru activare. Dezinstalarea unui TSR înseamnă eliberarea memoriei ocupate la instalare. Trebuie remarcat că dezinstalarea trebuie făcută de către un alt program decât cel rezident (eventual o altă instanță a aceluiași program).

Evitarea instalării repetate

Programul TSR este memorat într-un fișier executabil. Nimic nu ne împiedică să lansăm în execuție acest program de mai multe ori. Totuși, nu se poate accepta ca un TSR să fie instalat de mai multe ori, deoarece s-ar consuma în mod inutil memorie. Pe de altă parte, secvența de activare din Figura 8.8 ar fi compromisă (care instanță se va activa ?)

Un program TSR trebuie, deci, să testeze dacă nu este instalat deja în memorie și să refuze instalarea în acest caz.

Testarea se face definind o secvență particulară de caractere (o semnătură) la un offset fix în cadrul unicului segment care compune programul TSR. Identificarea locului în care programul TSR este eventual rezident se poate face numai prin examinarea vectorului de întrerupere care declanșează activarea (uzual, INT 9). Citind acest vector, identificăm adresa de segment a rutinei de tratare, adică adresa de segment a programului TSR care este eventual instalat. Comparând semnătura programului curent cu semnătura din segmentul indicat de vectorul INT 9, se determină dacă TSR-ul este deja instalat (vezi Figura 8.9).

Această tehnică este posibilă numai dacă TSR-ul deja instalat este ultimul din lanțul de TSR-uri care au modificat vectorul de întrerupere pe nivelul 9.

O altă soluție, mai generală, dar mai lentă, este parcurgerea tuturor blocurilor de memorie alocate de către sistem. Aceste blocuri sunt structuri de date specifice DOS care conțin informații depre deținătorul blocului, deci cărui modul de program i-a fost alocat blocul respectiv. Astfel, se caută de fapt în memorie numele fișierului executabil care conține programul TSR. În acest capitol, vom utiliza prima tehnică de identificare.

Instalarea trebuie să salveze o serie de date necesare procesului de activare, cum ar fi adresa de PSP și a zonei DTA a programului TSR, adresa stivei programului TSR etc. De asemenea, trebuie rezervat spațiu pentru stiva TSR, pentru salvarea ecranului, pentru copiile vectorilor de întrerupere modificați etc.

Instalarea TSR-ului mai presupune și modificarea vectorilor de întrerupere 9, 10H, 13H etc. cu adresele propriilor rutine de tratare. În final, se calculează dimensiunea spațiului de memorie care trebuie să rămână rezident și se încheie execuția cu apelul funcției DOS 31H.

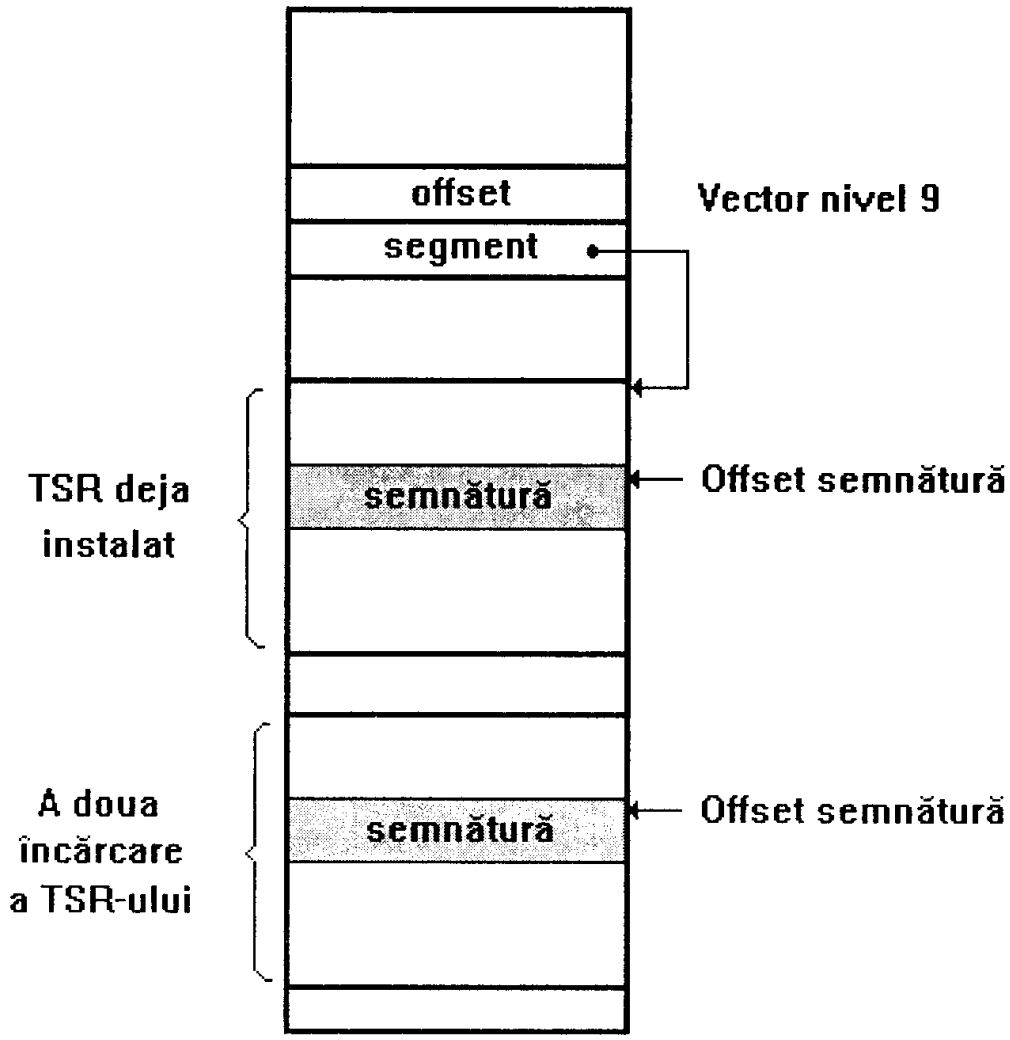


Figura 8.9 Tehnica de testare a instalării

Dezinstalarea TSR-urilor

La identificarea situației de TSR deja instalat, multe programe se mulțumesc să afișeze la consolă un mesaj de genul:

```
Program XXXXX already installed !
```

și apoi să-și încheie execuția în mod normal.

O variantă mai evoluată este ca, la a doua invocare a programului TSR, să se realizeze dezinstalarea primei instanțe, după care a doua instanță să se încheie cu ieșire normală în DOS.

Dezinstalarea trebuie să elibereze spațiul de memorie alocat primei instanțe. Trebuie mai întâi eliberat blocul de environment (mediu) al TSR-ului instalat. Adresa blocului de environment se găsește la un offset fix în cadrul PSP-ului (vezi 8.3). La instalare, este necesară memorarea adresei blocului PSP într-o variabilă aflată la un offset fix. După ce s-a eliberat blocul de environment, se

eliberează memoria propriu-zisă cu funcția 49H și se încheie execuția cu funcția DOS 4CH. Figura 8.10 ilustrează acest proces.

De observat că, funcție de testul asupra instalării TSR-ului, se va face instalarea instanței curente sau dezinstalarea instanței anterior instalate. Tot la dezinstalare se refac vectorii 9, 10H, 13H și 28H, care au fost modificați la instalare.

8.6.3 Șablon de dezvoltare pentru programe TSR

Următorul program reprezintă un șablon de dezvoltare pentru aplicațiile TSR uzuale, fiecare acțiune fiind comentată intensiv. Trebuie observat accesul la buffer-ul tastaturii, realizat prin pointerii de la adresele absolute 400:1AH și 400:1CH, care indică cele două capete ale buffer-ului. Buffer-ul tastaturii se află în spațiul de adrese 400:1EH-400:3CH, fiind organizat circular. Buffer-ul este vid când cei doi pointeri sunt egali și este plin când pointerul de introducere este cu o poziție (2 octeți) în urma pointerului de extragere. Capacitatea logică este de 30 de octeți (15 taste), iar cea fizică de 32 de octeți (16 taste).

Acest model de program realizează doar ștergerea ecranului, afișarea unui mesaj și așteptarea apăsării unei taste oarecare. Pentru afișări la consolă se folosesc direct funcții DOS, ceea ce explică terminatorul '$' pentru șiruri de caractere. Tasta de activare este F1. La o a doua execuție, programul se dezinstalează.

Se observă și funcțiile DOS sau BIOS pentru citirea sau poziționarea diverselor date de sistem.

```
bios segment at 40h             ; Segment date BIOS
      org   1AH
      b_out dw    ?             ; Indicator extragere
                                ; buffer BIOS
      b_in  dw    ?             ; Indicator introducere
                                ; buffer BIOS
bios ends

cseg segment
assume cs:cseg
MONO          equ   0
F1            equ   3B00H       ; 59 in zecimal = cod F1
HOTKEY        equ   F1          ; Tasta activare

ident         db    '_IDENTIF_'
lng           equ   $ - ident
mes_act       db    'TSR activ: Apasati orice tasta '
              db    'pentru a reveni'
              db    7, '$'
mes_inst      db    ' TSR instalat: Activare cu F1', '$'
mes_dezinst db      ' TSR dezinstalat', '$'
;
;     Spatiu pentru memorat vectorii vechi de intrerupere
```

```
;
      old9  label dword
            old9_off    dw    ?
            old9_seg    dw    ?
      old10 label dword
            old10_off   dw    ?
            old10_seg   dw    ?
      old13 label dword
            old13_off   dw    ?
            old13_seg   dw    ?
      old28 label dword
            old28_off   dw    ?
            old28_seg   dw    ?
;
;     Spatiu pentru adresa flagului INDOS
;
      flag_dos label dword
      indos_off           dw    ?
      indos_seg           dw    ?
;
      flag_tsr            dw    0             ; Flag TSR activ
      flag_bios           dw    0             ; Flag BIOS activ
      cerere              db    0             ; Memoreaza cerere
                                              ; viitoare
      cod_scan            dw    ?             ; Cod tasta activare
      ss_int              dw    ?             ; Salvare registre
                                              ; stiva
      sp_int              dw    ?             ; program intrerupt
      ss_tsr              dw    cseg          ; Stiva locala TSR
                          dw    256 dup (?)
      sp_tsr              dw    sp_tsr
      buff_video          db    4000 dup (?); Buffer salvare ecran
      salv_cursor         dw    ?             ; Salvare cursor
      dta                 db    43 dup (?)    ; DTA local TSR
      psp_tsr             dw    ?             ; Adresa PSP TSR
      dta_tsr             dd    dta           ; Adresa DTA TSR
;
      psp_int             dw    ?             ; Salvare PSP si
      dta_int             dd    ?             ; DTA program intrerupt
;
;     Noile rutine de tratare intreruperi 9, 10H, 13H, 28H
;
new10 proc far
      mov   flag_bios, 1
      pushf
      call  dword ptr cs:old10
      mov   flag_bios, 0
      retf  2
new10 endp
new13 proc  far
      mov   flag_bios, 1
      pushf
      call  dword ptr cs:old13
      mov   flag_bios, 0
      retf  2
```

```
new13 endp

new9  proc  far
      assume ds:bios
      pushf
      call  cs:old9                    ; Apel rutina veche
      cli                              ; Dezactivare
                                       ; intreruperi
      cmp   cs:flag_tsr, 0             ; Test deja activ
      jne   new9_end
      push  ds                         ; Salvari
      push  bx                         ; registre
      push  ax
      mov   ax, bios
      mov   ds, ax                     ; Acces BIOS prin ds
      mov   bx,  b_out                 ; Indicator extr.
                                       ; buffer BIOS
new9_0:
      cmp   bx, b_in                   ; Este buffer-ul
                                       ; BIOS vid ?
      je    new9_1                     ; Da, return
      mov   ax, [bx]                   ; Nu e vid, iau codul
      mov   cs:cod_scan, ax            ; si il memorez
      cmp   ax, HOTKEY                 ; Este tasta
                                       ; de activare ?
      jnz   new9_1                     ; Nu, return
      add   bx, 2                      ; Incrementez (extrag
                                       ; din buffer)
      cmp   bx, 3EH                    ; Compar cu val.
                                       ; maxima + 2
      jne   new9_11                    ; Nu este = 3EH

      mov   bx,1eh                     ; Este 3EH si se aduce
                                       ; la val. initiala
new9_11:
      mov   b_out, bx                  ; Memorare indicator
                                       ; buffer BIOS
      jmp   new9_2
new9_1:
      pop   ax
      pop   bx
      pop   ds
      iret

assume      ds: NOTHING

new9_2:
      pop   ax
      pop   bx
      pop   ds

      cmp   cs:flag_bios, 0            ; Test operatii critice
      je    cont1
      mov   cs:cerere, 1               ; Marcheaza cerere
new9_end:
```

```
        iret                                    ; Revenire in programul
                                                ; intrerupt
cont1:
        push  ds
        push  bx
        lds   bx, cs:flag_dos                   ; Test INDOS
        cmp   byte ptr [bx], 0
        je    cont2
        mov   cs:cerere, 1                      ; Marcheaza cerere
        pop   bx
        pop   ds
        iret                                    ; Revenire in programul
                                                ; intrerupt
cont2:
        pop   bx
        pop   ds
        call  activ_tsr                         ; Apel functie TSR
        iret
new9    endp
;
new28 proc far
        pushf                                   ; Apel rutina
        call  dword ptr cs:old28                ; veche
        cli
        cmp   flag_tsr, 0                       ; Test TSR activ
        jz    cont3
        iret
cont3:
        cmp   flag_bios, 0                      ; Test operatii critice
        jz    cont4
cont5:
        ret   2
cont4:
        cmp   cerere, 0                         ; Este cerere ?
        jz    cont5
        mov   cerere,0
        call  near ptr activ_tsr                ; Satisface cererea
        ret   2
new28   endp
;
;       Rutina de baza care realizeaza activarea TSR-ului
;
activ_tsr proc near
        cli
        mov   flag_tsr, 1                       ; Marcheaza TSR activ
;
;       Comutarea contextului
;
        mov   sp_int, sp                        ; Salveaza stiva
        mov   sp, ss                            ; program intrerupt
        mov   ss_int, sp
        mov   sp, ss_tsr                        ; Comut pe stiva
                                                ; TSR-ului
        mov   ss, sp
        mov   sp, sp_tsr
```

```
;
        push    ax                              ; Salvari
        push    bx                              ; registre
        push    cx
        push    dx
        push    si
        push    di
        push    bp
        push    ds
        push    es
;
        mov     cx, 64                          ; Salvare 64
        mov     ax, ss_int                      ; de octeti
        mov     ds, ax                          ; din stiva
        mov     si, sp_int                      ; programului
        cld                                     ; intrerupt
iar1:
        push    [si]                            ; in stiva
        inc     si                              ; proprie
        inc     si
        loop    iar1
;
        mov     ah, 62H                         ; Citeste PSP-ul
        int     21h                             ; programului intrerupt
        mov     psp_int, bx                     ; si salveaza
;
        push    es
        mov     ah, 2FH                         ; Citeste DTA
        int     21h                             ; programului intrerupt
        mov     word ptr dta_int, bx            ; si
        mov     word ptr dta_int+2, es          ; salveaza
        pop     es
;
        mov     ah,50H                          ; Marcheaza PSP-ul
        mov     bx,psp_tsr                      ; TSR-ului
        int     21h                             ; ca fiind cel activ
;
        push    ds                              ; Comuta
        mov     ah,1ah                          ; pe zona DTA
        lds     dx,dta_tsr                      ; a TSR-ului
        int     21h
        pop     ds
;
        sti
        cmp     cod_scan, HOTKEY                ; Test tasta activare
        jz      e1                              ; sau nu
        jmp     end_tsr
e1:
        mov     ah, 3
        mov     bh, 0                           ; Citire
        int     10H                             ; pozitie
        mov     salv_cursor, dx                 ; cursor si salvare
;
IF MONO
        mov     ax, 0B000H                      ; Adresa ecran monocrom
```

```
ELSE
        mov     ax, 0B800H                      ; Adresa ecran color
ENDIF
;
        push    ds
        push    es
        mov     ds, ax                          ; Sursa = B000:0
                                                ; (B800:0)
        mov     ax, cs
        mov     es, ax                          ; Destinatie =
                                                ; cs:buff_video
        cld
        xor     si, si
        mov     di, offset buff_video
        mov     cx, 4000                        ; Copiere 80*25*2
        rep     movsb                           ; octeti
        pop     es
        pop     ds
;
        mov     bh, 7                           ; Stergere
        mov     ax, 600H                        ; ecran
        mov     cx, 0                           ; De la linia 0,
                                                ; coloana 0
        mov     dx, 24*256 + 79                 ; pana la linia 24,
                                                ; coloana 79
        int     10H
;
;       Aici se pune codul propriu-zis pentru functia TSR-ului
;       in cazul de fata, se tipareste un mesaj si se
;       asteapta apasarea unei taste
;
        mov     bh, 0                           ; Pozitionare cursor in
        mov     dx, 010CH                       ; linia 1, coloana 0CH
        mov     ah, 2
        int     10H
;
        mov     ax, cs
        mov     ds, ax                          ; DS = segment curent
        mov     ah, 9
        mov     dx, offset mes_act              ; Mesaj TSR activ
        int     21H
        mov     ah, 0                           ; Citire caracter de la
                                                ; tastatura
        int     16H
;
;       Refacerea contextului
;
        mov     dx, salv_cursor                 ; Pozitionare cursor
        mov     bh, 0
        mov     ah, 2
        int     10H
;
        mov     ax, cs                          ; Refacere
        mov     ds, ax                          ; ecran
        mov     si, offset buff_video
```

```
;
IF MONO
      mov   ax, 0B000H
ELSE
      mov   ax, 0B800H
ENDIF
;
      mov   es, ax
      xor   di, di
      cld
      mov   cx, 4000
      rep   movsb
;
end_tsr:
      cli
      mov   ah, 50H                    ; Comutare pe PSP-ul
      mov   bx, psp_int                ; programului intrerupt
      int   21H
;
      push  ds
      mov   ah, 1aH                    ; Comutare pe DTA a
      lds   dx, dta_int                ; programului intrerupt
      int   21H
      pop   ds
;
      mov   cx, 64                     ; Refacere continut
                                       ; stiva
      mov   ax, ss_int                 ; program
      mov   ds, ax                     ; intrerupt
      mov   si, sp_int                 ; din stiva
      add   si, 128                    ; curenta
iar2:
      dec   si
      dec   si
      pop   [si]
      loop  iar2
;
      pop   es                         ; Refacere
      pop   ds                         ; registre
      pop   bp                         ; din stiva
      pop   di                         ; curenta (TSR)
      pop   si
      pop   dx
      pop   cx
      pop   bx
      pop   ax
;
      mov   sp, ss_int                 ; Refacere registre
                                       ; stiva
      mov   ss, sp                     ; ale programului
      mov   sp, sp_int                 ; intrerupt
;
      mov   flag_tsr, 0                ; Marcare TSR inactiv
      sti
      ret
```

```
activ_tsr endp
;
;       Rutina urmatoare reface vectorii de intrerupere
;       si elibereaza memoria alocata pentru programul TSR
;
dezinst     proc near
;
;       in ES : segmentul programului rezident
;
        push  ds
        push  es
        cli
;
;       Refacere vectori de intrerupere 9, 10H, 13H, 28H
;
        mov   ax, 2509H                     ; Vector 9
        mov   ds, es:old9_seg
        mov   dx, es:old9_off
        int   21H
;
        mov   ax, 2510H                     ; Vector 10H
        mov   ds, es:old10_seg
        mov   dx, es:old10_off
        int   21H
;
        mov   ax, 2513H                     ; Vector 13H
        mov   ds, es:old13_seg
        mov   dx, es:old13_off
        int   21H
;
        mov   ax, 2528H                     ; Vector 28H
        mov   ds, es:old28_seg
        mov   dx, es:old28_off
        int   21H
        sti
;
;       Prima data eliberam blocul de environment, a carui adresa
;       de segment se gaseste la offset-ul 2CH in PSP
;
        mov   bx, es:psp_tsr
        mov   es, bx                        ; Adresa PSP
        mov   es, es:[2ch]                  ; ES = adresa bloc
                                            ; environment
        mov   ah, 49H                       ; Eliberare memorie
                                            ; alocata
        int   21H                           ; la instalare
;
;       Acum eliberam memoria detinuta de programul
;       TSR, incepand de la PSP
;
        mov   es, bx                        ; ES = PSP-ul TSR-ului
        mov   ah, 49H                       ; Eliberare
        int   21H
;
        pop   es
```

```
        pop     ds
        ret                                     ; Revenire
dezinst endp
;
sfirsit label byte                              ; Eticheta pentru
                                                ; a pastra rezident
                                                ; numai pana aici
;
;       Aici e punctul de start al programului
;
start:
        mov     ax, cseg                        ; Initializari registre
        mov     ds, ax                          ; de
        mov     ss, ax                          ; segment
        mov     sp, sp_tsr                      ; si stiva
;
;       Testam daca TSR-ul este deja instalat
;
        mov     ax, 3509H                       ; Citire vector de
        int     21H                             ; intrerupere 9
                                                ; ES = segment program
                                                ; instalat
                                                ; deja (eventual)
        mov     ax, cs
        mov     ds, ax                          ; DS = segment  program
                                                ; curent
        xor     si, si                          ; Offset-uri 0
        mov     di, si
        mov     cx, lng                         ; Lungime mesaj
        repz    cmpsb                           ; Comparam ds:[si]
                                                ; cu es:[di]
        jnz     not_inst                        ; Nu e instalat, salt
;
        call    dezinst                         ; Dezinstalare, apoi
        mov     dx, offset mes_dezinst          ; mesaj dezinstalare
        mov     ah, 9
        int     21H
;
        mov     ax, 4c00H                       ; Iesire normala
        int     21H

not_inst:
;
;       Instalare TSR
;
        mov     ah, 62H                         ; Citire PSP si salvare
        int     21H                             ; pentru comutare
                                                ; context
        mov     psp_tsr, bx                     ; la activari viitoare
;
        mov     ah, 1aH                         ; Similar pentru DTA
        lds     dx, dta_tsr
        int     21H
;
;       Citiri si salvari vectori vechi
```

```
;       (nivelele 9, 10H, 13H, 28H)
;
        mov     ax, 3509H
        int     21H
        mov     cs:old9_off, bx
        mov     cs:old9_seg, es                 ; Vector 9
        mov     ax, 3510H
        int     21H
        mov     cs:old10_off, bx
        mov     cs:old10_seg, es                ; Vector 10H
        mov     ax,3513H
        int     21H
        mov     cs:old13_off, bx
        mov     cs:old13_seg, es                ; Vector 13H
        mov     ax, 3528H
        int     21H
        mov     cs:old28_off, bx
        mov     cs:old28_seg, es                ; Vector 28H
        mov     ax, cs
        mov     ds, ax                          ; DS = segmentul de cod
;
        push    ds
        push    es
;
;       Pozitionare vectori noi
;
        mov     ax, 2509H
        mov     dx, offset new9
        int     21H                             ; Vector 9 nou
;
        mov     ax, 2510H
        mov     dx, offset new10
        int     21H                             ; Vector 10H nou
;
        mov     ax, 2513H
        mov     dx, offset new13
        int     21H                             ; Vector 13H nou
;
        mov     ax, 2528H
        mov     dx, offset new28
        int     21H                             ; Vector 28H nou
;
;       Citire si salvare adresa flag INDOS
;
        mov     ax, 3400H
        int     21H
        mov     cs:indos_off, bx
        mov     cs:indos_seg, es
;
        pop     es
        pop     ds
;
        mov     ah, 9
        mov     dx, offset mes_inst             ; Mesaj instalare
        int     21H
```

```
;
;       Calcul al necesarului de memorie (inclusiv PSP-ul)
;
        mov     dx, offset sfirsit + 100h
        mov     cl, 4                           ; Spatiul se specifica
        shr     dx, cl                          ; in paragrafe de
                                                ; cate 16 octeti
        inc     dx
        mov     ax, 3100H                       ; Terminare cu ramanere
                                                ; in memorie
        int     21H

cseg    ends
end     start
```

8.7 Funcții de intrare/ieșire orientate pe handlere

Handlerele sunt variabile întregi mai mari sau egale cu zero, întoarse de sistemul DOS sau predefinite, care identifică în mod univoc fișiere disc sau dispozitive. Funcțiile orientate pe handlere au generalitate maximă și permit schimbarea dispozitivelor de intrare și/sau ieșire asociate unui program într-o manieră foarte comodă.

Sunt predefinite următoarele handlere:

- 0 - numit și **stdin** (**Standard Input - Intrare standard**), uzual asignat la dispozitivul consolă;
- 1- numit și **stdout** (**Standard Output - Ieșire standard**), uzual asignat la dispozitivul consolă;
- 2 - numit și **stderr** (**Standard Error - Ieșire standard pentru mesaje de eroare**), totdeauna asignat la consolă;
- 3 - numit și **stdaux** (**Standard Auxilliary - Dispozitiv auxiliar**), uzual asignat la dispozitivul COM1;
- 4 - numit și **stdprn** (**Standard Printer - Imprimantă standard**), uzual asignat la dispozitivul LPT1.

Funcțiile orientate pe handlere pot accesa orice subdirector al discurilor, pornind de la rădăcină sau de la directorul curent.

Intrarea și ieșirea standard asociate unui program executabil pot fi redirectate din linia de comandă, cu ajutorul caracterelor < (redirectare sttdin) și > (redirectare stdout). Dacă un program executabil **nume.exe** utilizează funcții de intrare/ieșire orientate pe handlere, atunci o lansare în execuție de forma:

```
C:\> nume < file1
```

va face ca toate citirile care au loc de la consolă să fie înlocuite cu citiri din fișierul **file1**.

Similar, o lansare în execuție de forma:

```
C:\> nume > file2
```

va face ca toate afișările care au loc la consolă să fie înlocuite cu scrieri în fișierul **file2**.

În fine, o linie de comandă de forma:

```
C:\> nume < file1 > file2
```

va redirecta atât stdin, cât și stdout.

Interpretorul de comenzi permite și un simulacru de mecanism **pipe**, prin care ieșirea standard a unui program **nume1** este redirectată către intrarea standard a altui program **nume2**:

```
C:\> nume1 | nume2
```

Cum sistemul DOS nu poate executa programe în mod concurent, linia de mai sus este echivalentă cu secvența:

```
C:\> nume1 > temp
C:\> nume2 < temp
C:\> del temp
```

În continuare, sunt descrise principalele funcții de intrare/ieșire orientate pe handlere și parametrii lor de intrare și de ieșire.

- **Create File (Creează sau recreează fișier)**
  - Parametri de intrare:

    AH = 3CH (cod funcție)

    CX = atribute (Read Only, Hidden, System etc.)

    DS:DX = adresă șir de caractere terminat cu 0 (calea fișierului)
  - Parametri de ieșire:

    CF = 0, AX = handler (operație încheiată cu succes)

    CF = 1, AX = cod de eroare (eșec)
  - Descriere:

    Se creează fișierul cu numele specificat; dacă fișierul există, conținutul vechi se pierde.
- **Open File (Deschide fișier)**
  - Parametri de intrare:

    AH = 3DH (cod funcție);

    AL = mod acces (0 = pentru citire, 1 = pentru scriere, 2 = punere la zi etc.

    DS:DX = adresă șir de caractere terminat cu 0 (calea fișierului);

- Parametri de ieșire:

  CF = 0, AX = handler (operație încheiată cu succes);

  CF = 1, AX = cod de eroare (eșec);

- Descriere:

  Se deschide fișierul cu numele specificat, pentru operația specificată; fișierul trebuie să existe; la deschidere pentru scriere, conținutul vechi se pierde.

- **Read File (Citește din fișier)**
  - Parametri de intrare:

    AH = 3FH (cod funcție);

    BX = handler (obținut la deschidere sau predefinit);

    DS:DX = adresă buffer de citire (dimensionat corespunzător);

    CX = număr de octeți cerut pentru citire;

  - Parametri de ieșire:

    CF = 0, AX = număr de octeți citiți efectiv (operație încheiată cu succes);

    CF = 1, AX = cod de eroare (eșec);

  - Descriere:

    Se încearcă citirea a CX octeți din fișier, din poziția curentă; poziția este avansată automat; dacă de la poziția curentă până la sfârșitul fișierului sunt mai puțin de CX octeți, se citesc câți octeți există (posibil 0); dacă AX = 0, s-a ajuns la EOF înainte de apelul funcției, iar dacă AX < CX, s-a ajuns la EOF în decursul execuției funcției; ambele situații sunt corecte.

- **Write File (Scrie în fișier)**
  - Parametri de intrare:

    AH = 40H (cod funcție);

    BX = handler (obținut la deschidere sau predefinit);

    DS:DX = adresă buffer de scriere;

    CX = număr de octeți cerut pentru scriere;

  - Parametri de ieșire:

    CF = 0, AX = număr de octeți scriși efectiv (AX diferă de CX numai în situații de eroare);

    CF = 1, AX = cod de eroare (eșec);

  - Descriere:

Se încearcă scrierea a CX octeți în fișier, din poziția curentă; poziția este avansată automat; dacă CX < AX, este vorba de o situație de eroare.

- **Close File (Închide fișier)**
  - ♦ Parametri de intrare:

    AH = 3EH (cod funcție);

    BX = handler (obținut la deschidere);
  - ♦ Parametri de ieșire:

    CF = 0 (operație încheiată cu succes);

    CF = 1, AX = cod de eroare (eșec);
  - ♦ Descriere:

    Se închide fișierul, adică se taie legătura logică dintre handler și fișier, stabilită la deschidere; operația este obligatorie, în special la fișierele deschise pentru scriere, deoarece acum se scriu bufferele interne ale sistemului DOS în fișier.
- **Delete File (Șterge fișier)**
  - ♦ Parametri de intrare:

    AH = 41H (cod funcție);

    DS:DX = adresă șir de caractere terminat cu 0 (calea fișierului);
  - ♦ Parametri de ieșire:

    CF = 0 (operație încheiată cu succes);

    CF = 1, AX = cod de eroare (eșec);
  - ♦ Descriere:

    Se șterge fișierul cu numele specificat.
- **Duplicate Handle (Duplică handler)**
  - ♦ Parametri de intrare:

    AH = 45H (cod funcție);

    BX = handler (obținut la deschidere sau predefinit);
  - ♦ Parametri de ieșire:

    CF = 0., AX = handler duplicat (operație încheiată cu succes);

    CF = 1, AX = cod de eroare (eșec);
  - ♦ Descriere:

    Se obține o copie a handlerului asociat fișierului; operația este necesară înainte de redirectarea unor fișiere sau dispozitive, în situația în care dorim și revenirea la asocierea inițială.

- **Redirect Handle (Redirectează handler)**
  - Parametri de intrare:

    AH = 46H (cod funcție);

    CX = handler care trebuie redirectat;

    BX = handler (obținut la deschidere sau predefinit);
  - Parametri de ieșire:

    CF = 0., AX = handler duplicat (operație încheiată cu succes);

    CF = 1, AX = cod de eroare (eșec);
  - Descriere:

    Se obține o copie a handlerului asociat fișierului; operația este necesară înainte de redirectarea unor fișiere sau dispozitive, în situația în care dorim revenirea la asocierea inițială.
- **Seek File (Poziționează indicatorul fișierului)**
  - Parametri de intrare:

    AH = 42H (cod funcție);

    BX = handler (obținut la deschidere);

    CX:DX = poziția cerută în fișier (întreg pe 32 de biți, CX = partea high, DX = partea low);

    AL = punctul de referință (0 = față de începutul fișierului, 1 față de poziția curentă, 2 = față de sfârșitul fișierului);
  - Parametri de ieșire:

    CF = 0, DX:AX = noua poziție în fișier (operație încheiată cu succes);

    CF = 1, AX = cod de eroare (eșec);
  - Descriere:

    Se încearcă citirea a CX octeți din fișier, din poziția curentă; poziția este avansată automat; dacă de la poziția curentă până la sfârșitul fișierului sunt mai puțin de CX octeți, se citesc câți octeți există (posibil 0); dacă AX = 0, s-a ajuns la EOF înainte de apelul funcției, iar dacă AX < CX, s-a ajuns la EOF în decursul execuției funcției; ambele situații sunt corecte.
- **Find First (Identifică primul fișier dintr-un nume generic cu * și/sau ?)**
  - Parametri de intrare:

    AH = 4EH (cod funcție);

    DS:DX = adresă șir de caractere terminat cu 0 (poate conține * și ?);

    CX = atribute (0 = normal);

- Parametri de ieșire:

  CF = 0, zona DTA conține informații despre fișierul identificat (operație încheiată cu succes);

  CF = 1, AX = cod eroare (nu s-a identificat fișier);

- Descriere:

  Se caută primul fișier care verifică numele generic specificat. La offset-ul 1EH în zona DTA se găsește numele acestui fișier; este necesară obținerea adresei zonei DTA în prealabil (necesară la funcția următoare).

- **Find Next (Identifică următorul fișier dintr-un nume generic)**
  - Parametri de intrare:

    AH = 4FH (cod funcție);

    DS:DX = adresa zonei DTA sau a unei copii a acestei zone;

  - Parametri de ieșire:

    CF = 0, zona DTA conține informații despre următorul fișier identificat (operație încheiată cu succes);

    CF = 1, AX = cod eroare (nu s-a identificat fișier);

  - Descriere:

    Se caută următorul fișier care verifică numele generic specificat la un apel anterior al funcției Find First. La offset-ul 1EH în zona DTA se găsește numele acestui fișier; este necesară obținerea adresei zonei DTA în prealabil (necesară la funcția următoare). În general, această funcție se apelează ciclic, până când se obține CF = 1 (nu mai sunt fișiere care să verifice numele generice).

Funcțiile de duplicare și de redirectare sunt utilizate de interpretorul de comenzi, pentru redirectarea din linia de comandă a intrării și a ieșirii standard. Redirectări asemănătoare pot fi făcute și din programele utilizator. Să considerăm următorul program, care execută următoarele acțiuni:

- scrie un mesaj la ieșirea standard;
- creează un fișier disc;
- duplică handlerul 1 (ieșirea standard);
- redirectează ieșirea standard către fișierul anterior creat;
- scrie un mesaj la ieșirea standard (scrierea se va face, de fapt, în fișier);
- redirectează ieșirea standard la handlerul duplicat (revenire la situația inițială);
- scrie un mesaj la ieșirea standard.

Programul este listat în continuare:

```
.model large
include io.h
.data
      nume  db 'fis.dat', 0
      h1    dw ?
      h2    dw ?
      mes_1 db 'Acest mesaj se va tipari la consola', cr, lf
      lng_1 dw $ - mes_1
      mes_2 db 'Acest mesaj se va tipari in fis.dat', cr, lf
      lng_2 dw $ - mes_2
      mes_3 db 'Acest mesaj se va tipari din nou la consola', cr, lf
      lng_3 dw $ - mes_3
 .stack 1024
 .code
 start:
      init_ds_es
;
      mov   ah, 40H                 ; Scriere la
      mov   bx, 1                   ; standard output
      lea   dx, mes_1
      mov   cx, lng_1
      int   21H
;
      mov   ah, 45H                 ; Duplicare
      mov   bx, 1                   ; standard output
      int   21H
      mov   h2, ax                  ; Handler duplicat in h2
;
      mov   ah, 3CH                 ; Deschidere 'fis.dat'
      mov   cx, 0                   ; pentru scriere
      lea   dx, nume
      int   21H
      mov   h1, ax                  ; Handler in h1
;
      mov   ah, 46H                 ; Redirecteaza
      mov   cx, 1                   ; standard output
      mov   bx, h1                  ; la h1, adica
                                    ; la 'fis.dat'
      int   21H
;
      mov   ah, 40H                 ; Scriere la
      mov   bx, 1                   ; standard output
                                    ; redirectat
      lea   dx, mes_2
      mov   cx, lng_2
      int   21H
;
      mov   ah, 46H                 ; Redirecteaza
      mov   cx, 1                   ; standard output
      mov   bx, h2                  ; la h2, adica la
                                    ; cel original
      int   21H
;
      mov   ah, 40H                 ; Scriere la
```

```
        mov     bx, 1                   ; standard output
                                        ; original
        lea     dx, mes_3
        mov     cx, lng_3
        int     21H
;
        exit_dos
end     start
```

Mesajele mes_1 și mes_3 vor fi afișate pe ecran, iar mesajul mes_2 va fi scris în fișierul fis.dat. Programul de mai sus (să presupunem că se numește redi.exe) se poate acum rula și cu redirectare din linia de comandă:

```
C:\> redi > fis_nou.dat
```

ceea ce va duce la scrierea mesajelor mes_1 și mes_2 în fișierul fis_nou.dat. Se observă că redirectările efectuate prin program se referă la asignarea curentă a dispozitivului respectiv, nu la asignarea implicită.

O problemă importantă la scrierea fișierelor executabile este accesul la parametrii din linia de comandă. La descrierea blocului PSP, am văzut că în PSP există un contor al caracterelor din linia de comandă, iar „coada" propriu-zisă a comenzii se găsește începând de la offset-ul 81H.

Să considerăm un program care acceptă în linia de comandă un nume generic de fișier (cu '*' sau '?') și afișează la consolă toate fișierele care corespund numelui generic. Accesul la fișiere se va face prin funcțiile Find First și Find Next, iar parametrii din linia de comandă se citesc din PSP.

Se pune problema corectitudinii numărului de parametri (cuvinte). Vom utiliza procedura count, dezvoltată în subcapitolul 8.9, care primește un șir de caractere și întoarce numărul de cuvinte din acel șir. Este necesară o ușoară modificare, în sensul de a accepta ca delimitatori de cuvinte numai spațiul și caracterul Tab. Pentru apelul funcțiilor DOS, se utilizează macroinstrucțiunile definite în fișierul io.h (descris în Anexa A).

Implementarea programului este următoarea:

```
.model      large
include io.h
        extrn count : far
.stack      512
;
;       Zona proprie DTA
;
dta     struc
                    db      30      dup ()
        f_name      db      13      dup (); Nume fisier
                                            ; identificat
dta     ends
;
        stdout      equ     1
```

```
;
.data
      file_struc  dta    <>
      buf_name    db     128    dup (0)
      gen_name    db     80     dup (0)
      f_hand             dw     ?
      nr_bytes    dw     ?
      buf         db     4096   dup (?)
      dim_buf            equ    $ - buf
      help_err    db     cr, lf , 'Sintaxa este: dump <file>',0
      mess_no_file       db     cr, lf, 'dump: Nu exista fisier: ',0
      open_err    db     cr, lf, 'dump: Eroare deschidere fisier ',0
      read_err    db     cr, lf, 'dump: Eroare citire fisier ',0
.code
start:
      cld
      push  ds                          ; Avem nevoie de
                                        ; adresa PSP
      pop   ax
      init_ds_es
      mov   si, 80H                     ; Offset contor
                                        ; caractere
      lea   di, buf_name
      push  ds
      mov   ds, ax
      mov   cl, [si]
      xor   ch,ch                       ; CX = numar caractere, fara
                                        ; CR, LF
      inc   si                          ; Primul caracter
      jcxz  help_ds                     ; Daca CX = 0, mesaj de help
;
      push  cx
      rep   movsb                       ; Transfer din PSP
                                        ; in buf_name
      mov   byte ptr es:[di], 0         ; Terminator sir
      pop   cx
      pop   ds
      lea   si, buf_name
ch_nxt:
      cmp   byte ptr [si], ' '          ; Sarim peste spatiile albe
      jz    skip
      cmp   byte ptr [si], tab          ; si peste Tab-uri
      jz    skip
      jmp   first_ch
skip:
      inc   si                          ; De la cap la coada
      dec   cx                          ; si contorizam
      jmp   ch_nxt
;
first_ch:
      lea   di, gen_name                ; Transfer in gen_name
      jcxz  help                        ; Daca acum CX = 0, nu
                                        ; exista parametri
      rep   movsb                       ; Abia acum transfer
ch_last:
```

```
        dec     di                          ; Pe ultimul caracter
        cmp     byte ptr [di], ' '          ; Eliminam spatiile albe
                                            ; si de la sfarsit
        jz      skip1
        cmp     byte ptr [di], tab          ; La fel cu Tab-urile
        jz      skip1
        jmp     last_char
skip1:
        dec     di                          ; De la coada la cap
        dec     cx                          ; si contorizam
        jmp     ch_last
last_char:
        inc     di                          ; Am decrementat o data
                                            ; in plus
        mov     byte ptr [di], 0            ; Punem terminator
        lea     si, gen_name                ; Adresa sir curatat
                                            ; de spatii albe
                                            ; initiale si finale
        call    far ptr count               ; Numaram cuvintele
        cmp     ax, 1                       ; AX = numarul de cuvinte
        je      ok                          ; Daca e unul singur, e bine
        jmp     help                        ; Altfel, mesaj help
;
help_ds:
        pop     ds                          ; Help cu POP DS
help:
        puts    help_err                    ; Mesaj de asistenta
        exit_dos                            ; si iesire in DOS
;
ok:
        set_dta             file_struc      ; Setam DTA pe zona noastra
        find_first  gen_name                ; Apel Find First
        jnc     rel                         ; S-a gasit ceva
        jmp     gata_1                      ; Nu s-a gasit nimic
rel:
        o_read      file_struc.f_name, f_hand
                                            ; Deschide fisierul cu numele
                                            ; raportat de Find First in DTA
        jnc     no_err                      ; Test eroare
        puts    open_err                    ; Mesaj de eroare
        puts    file_struc.f_name           ; Afisare nume fisier
        putsi   <cr, lf>                    ; CR, LF
        jmp     next                        ; Si trecem la urmatorul
                                            ; fisier
no_err:
        putsi   <cr, lf, 'Fisier: '>        ; Nume
        puts    file_struc.f_name           ; fisier identificat
        putsi   <cr, lf, cr ,lf>
iar:
        f_read f_hand, buf, dim_buf
                                            ; Citire dim_buf
                                            ; octeti din fisier
                                            ; in bufferul buf
        jc      error                       ; Test eroare
        test    ax, ax                      ; Test sfarsit de fisier
```

```
                                        ; (s-au citit
                                        ; 0 octeti)
        jz      next                    ; Salt la urmatorul fisier
        mov     nr_bytes, ax            ; Memoram cat s-a citit
                                        ; de fapt
        f_write stdout, buf, nr_bytes
                                        ; Afisam ce s-a citit
        cmp     nr_bytes, dim_buf       ; Daca s-au citit mai
                                        ; putini octeti
        jb      close                   ; decat am cerut, salt
                                        ; la inchidere
        jmp     iar                     ; Altfel, reluam
                                        ; bucla de citire
close:
        f_close f_hand                  ; inchidere fisier
next:
        find_next file_struc            ; Urmatorul fisier
        jc      gata                    ; CF = 1 : nu mai sunt
        jmp     rel
error:
        puts    read_err                ; Mesaj de eroare
        puts    file_struc.f_name       ; Nume fisier
        putsi   <cr, lf>
        jmp     next                    ; Urmatorul
gata_1:
        puts    mess_no_file            ; Nu exista nici un
                                        ; fisier cu numele dat
        puts    gen_name
        putsi   <cr, lf>
gata:
        exit_dos                        ; Iesire in DOS
end     _main
```

Programul accesează PSP-ul, copiind „coada" comenzii într-un buffer propriu. Se sare apoi peste spațiile albe de la începutul șirului și se elimină și spațiile albe de la sfârșitul șirului (prin mutarea terminatorului mai la stânga).

Apoi se copiază șirul „curățat" în buffer-ul gen_name și se apelează procedura count, care întoarce numărul de cuvinte din șir. Dacă numărul de parametri este diferit de 1, se afișează un mesaj help. Altfel, se setează zona DTA pe o zonă proprie și se apelează Find First. La un offset specificat în zona DTA, se va găsi numele primului fișier (dacă CF = 0).

Se deschide fișierul pentru citire și se intră într-o buclă de citire-afișare, care se încheie dacă numărul de octeți citiți este strict mai mic decât cel cerut sau este zero.

Se apelează Find Next, cu parametrul zona proprie DTA, care întoarce numele următorului fișier identificat. Toată procedura se reia în buclă, până când Find Next raportează (prin CF = 0) că nu mai sunt fișiere care să corespundă numelui generic specificat.

Anexa A

Fișierul header io.h

Această anexă listează fișierul header io.h, care conține declarațiile de proceduri și definițiile de macroinstrucțiuni pentru operațiile de intrare/ieșire utilizate în această carte. Semnificația macroinstrucțiunilor a fost prezentată în subcapitolul 2.7. O parte din macroinstrucțiuni au fost explicate în Capitolul 5.

```
.xlist
        ;
        ;       Declaratii de proceduri externe (implementate in IO.ASM)

        extrn itoa_proc:far,   atoi_proc:far, puts_proc:far
        extrn gets_proc:far,   getc_proc:far, putc_proc:far
        extrn puti_proc:far,   putu_proc: far

        cr      equ     0DH                     ; Constante
        lf      equ     0AH                     ; simbolice
        tab     equ     09H                     ; uzuale
        ;
        ;       init_ds_es: Initializarea registrelor DS si ES
        ;
        init_ds_es  macro
                push    ax
                mov     ax, dgroup
                mov     ds, ax
                mov     es, ax
                pop     ax
        endm
        ;
        ;       exit_dos: Iesire in DOS
        ;
        exit_dos    macro
                mov     ax, 4C00H
                int     21H
        endm
        ;
        ;       itoa: Conversie intreg - ASCII
        ;
        itoa    macro dest, source
                push    ax
                push    di
                mov     ax, source                  ;; Numar intreg
                lea     di, dest                    ;; Adresa sir generat
                call    itoa_proc
                pop     di
                pop     ax
```

```
endm
;
;       atoi: Conversie ASCII - intreg

atoi    macro source
        push    si
        lea     si, source              ;; Adresa sir
        call    atoi_proc               ;; Rezultat in AX
        pop     si
endm
;
;       inputs: Citire sir cu numar maxim admis de caractere
;
inputs      macro dest, length
        push    di
        push    cx
        lea     di, dest                ;; Adresa sir
        mov     cx, length              ;; Lungime maxima
        call    gets_proc
        pop     cx
        pop     di
endm
;
;       getc: Citire caracter
;
getc    macro
        call    getc_proc               ;; Caracter in AL
endm
;
;       puti: Afisare intreg cu semn
;
puti    macro n
        push    n                       ;; intreg
        call    far ptr puti_proc
        add     sp, 2
endm
;
;       putu: Afisare intreg fara semn
;
putu    macro       n
        push    n                       ;; intreg
        call    far ptr putu_proc
        add     sp, 2
endm
;
;       geti: Citire întreg
;
geti    macro
        local buff
        .data
              buff  db    8 dup(0)
        .code
              inputs      buff, 7       ;; Cel mult sapte
                                        ;; cifre si semn
              atoi  buff                ;; ASCII - intreg
```

```
                                        ;; intreg in AX
endm
;
;       Citire intreg fara semn
;
getu    macro
        geti                            ;; E acelasi lucru
                                        ;; cu geti
endm
;
;       puts: Afisare sir
;
puts    macro x
        push    si
        lea     si, x                   ;; Adresa sir
        call    puts_proc
        pop     si
endm
;
;       putsi: Afisare sir imediat (din aceeasi linie)
;
putsi   macro x
        local string
        .data
                string      db    x, 0  ;; Sir definit local
        .code
                puts  string
endm
;
;       gets: Citeste un sir de la consola
gets    macro       x
        inputs      x, 80               ;; Cel mult
                                        ;; 80 de caractere
endm
;
;       putc: Afiseaza un caracter
;
putc    macro x
        push    ax
        mov     al, x
        call    far ptr putc_proc
        pop     ax
endm
;
;       getvec: Citeste vector de intrerupere
;
getvec      macro oldint, n
        push    bx
        push    ax
        push    es
        mov     al, n                   ;; Nivel
        mov     ah, 35H                 ;; Apel
        int     21H                     ;; functie DOS
        mov     word ptr oldint, bx     ;; Adresa depunere
        mov     word ptr oldint+2, es   ;; vector citit
```

```
        pop   es
        pop   ax
        pop   bx
endm
;
;       setvec: Modifica vector de intrerupere
;
setvec  macro newint, n
        push  ax
        push  ds
        push  dx
        mov   al, n                     ;; Nivel
        lds   dx, newint                ;; Adresa vector nou
        mov   ah, 25H                   ;; Apel
        int   21H                       ;; functie DOS
        pop   dx
        pop   ds
        pop   ax
endm
;
;       o_read: Deschide fisier pentru citire
;
o_read  macro fname, hand
        mov   ah, 3DH
        mov   al, 0C0h                  ;; Mod acces
        lea   dx, fname                 ;; Nume fisier
        int   21H
        mov   hand, ax                  ;; Depune handler
endm
;
;       o_write: Deschide fisier pentru scriere
;
o_write macro fname, hand
        mov   ah, 3DH
        mov   al, 0C1H                  ;; Mod acces
        lea   dx, fname                 ;; Nume fisier
        int   21H
        mov   hand, ax                  ;; Depune handler
endm
;
;       o_creat: Creeaza fisier
;
o_creat macro fname, hand
        mov   ah, 3CH
        mov   cx, 0                     ;; Atribute normale
        lea   dx, fname                 ;; Nume fisier
        int   21H
        mov   hand, ax                  ;; Depune handler
endm
;
;       f_read: Citeste date din fisier
;
f_read  macro hand, buf, nr
        mov   bx, hand                  ;; Handler
        mov   cx, nr                    ;; Numar octeti
```

```
            lea   dx, buf                   ;; Buffer (destinatie)
            mov   ah, 3FH
            int   21H
      endm
      ;
      ;     f_write: Scrie date in fisier
      ;
      f_write macro hand, buf, nr
            mov   bx, hand                  ;; Handler
            mov   cx, nr                    ;; Numar octeti
            lea   dx, buf                   ;; Buffer (sursa)
            mov   ah, 40H
            int   21H                       ;; AX = numar octeti
                                            ;; cititi efectiv
      endm
      ;
      ;     f_close: inchidere fisier
      ;
      f_close     macro hand
            mov   ah, 3EH
            mov   bx, hand                  ;; Handler
            int   21H
      endm
      ;
      ;     Fixeaza adresa zonei DTA
      ;
      set_dta     macro dta
            mov   ah, 1AH
            lea   dx, dta                   ;; Adresa zona DTA
                                            ;; (41 de octeti)
            int   21H
      endm
      ;
      ;     find_first: Determina primul fisier dintr-un nume
generic
      ;
      find_first  macro name
            mov   ah, 4EH
            lea   dx, name                  ;; Nume generic
            mov   cx, 0                     ;; Atribute normale
            int   21H
      endm
      ;
      ;     find_next: Determina urmatorul fisier dintr-un
      ;  nume generic
      ;
      find_next   macro dta_area
            mov   ah, 4FH
            lea   dx, dta_area              ;; Adresa zonei DTA
            int   21H
      endm
.sall
.list
```

Anexa B

Fișierul sursă IO.ASM

Această anexă cuprinde implementările procedurilor de intrare/ieșire utilizate în această carte. O parte din aceste proceduri au fost explicate în Capitolul 6 (Tehnici de programare în limbaj de asamblare). Toate procedurile sunt de tip far.

```
.model large
;
;       Declaratii de simboluri publice
;
        public      itoa_proc, atoi_proc, puts_proc, puti_proc
        public      gets_proc, getc_proc, putc_proc, putu_proc
        dosint      macro func
                    mov   ah, func          ;; Macro de apel
                                            ;; functii DOS
                    int   21H
        endm
        .code
        strrev      proc far
;
;       Schimba ordinea caracterelor din sirul indicat de DS:SI
;
              cmp   byte ptr [di], 0        ; Test terminator sir
              je    str_5

              push  di                      ; Salvari
              push  si
              push  ax
              mov   si, di                  ; Copie adresa sir
        str_1:
              mov   al, [si]                ; Determina sfarsit sir
              test  al, al
              jz    str_2
              inc   si                      ; Incrementeaza adresa
              jmp   str_1
        str_2:
              dec   si                      ; SI pe ultimul
                                            ; caracter util
        str_3:
              mov   al, [si]                ; Interschimba
              xchg  al, [di]                ; ultimul caracter
              mov   [si], al                ; cu primul
              inc   di
              dec   si
              cmp   di, si                  ; Pana cand
                                            ; pointerul crescator
```

```
            jb      str_3                   ; devine mai mare decat
                                            ; cel descrescator

            pop     ax                      ; Refaceri
            pop     si
            pop     di
        str_5:
            retf
        strrev      endp

        itoa_proc proc far
;
;       Converteste intregul cu semn din AX la sirul de caractere
;       de la adresa DS:DI
;
            push    ax                      ; Salvari
            push    bx
            push    cx
            push    dx
            push    di
;
            cmp     ax, 8000H               ; Caz special
            jne     itoa_1
            mov     byte ptr [di], '-'      ; care se
            mov     byte ptr [di+1], '3'    ; converteste
            mov     byte ptr [di+2], '2'    ; manual
            mov     byte ptr [di+3], '7'    ; deoarece NEG din
                                            ; 8000H da
            mov     byte ptr [di+4], '6'    ; tot 8000H (adica
                                            ; -32768)
            mov     byte ptr [di+5], '8'
            mov     byte ptr [di+6], 0
            jmp     itoa_2
        itoa_1:
            mov     cl, ' '                 ; Memoram semnul
            cmp     ax, 0
            jge     itoa_3
            mov     cl, '-'
            neg     ax                      ; Valoare absoluta
        itoa_3:
            mov     bx, 10                  ; impartitor
        itoa_4:
            mov     dx, 0                   ; impartire 32 de biti
            div     bx                      ; la 16 biti
            add     dl, '0'                 ; Cifra curenta
            mov     [di], dl                ; depusa in sir
            inc     di
            test    ax, ax                  ; Pana cand AX devine 0
            jnz     itoa_4
            cmp     cl, '-'                 ; Test semn
            jne     itoa_6
            mov     [di], cl                ; Pune minus
            inc     di
        itoa_6:
            mov     byte ptr [di], 0        ; Terminator
```

```
        pop   di                        ; Adresa de inceput
        call  far ptr strrev            ; Inverseaza sir
        push  di                        ; Pentru POP-urile
                                        ; care vin
itoa_2:
        pop   di                        ; Refaceri
        pop   dx
        pop   cx
        pop   bx
        pop   ax
        retf
itoa_proc endp

atoi_proc proc far
;
;       Converteste sirul de la adresa DS:SI la intreg in AX
;
        push  di                        ; Salvari
        push  si
        push  bx
        push  cx
        push  dx
atoi_1:
        cmp   byte ptr [si], ' '        ; Sarim peste
                                        ; blancurile initiale
        jne   atoi_2
        inc   si
        jmp   atoi_1
atoi_2:
        mov   di, 1                     ; Memorare semn  '+'
        cmp   byte ptr [si], '+'        ; Test semn
        je    atoi_3
        cmp   byte ptr [si], '-'
        jne   atoi_4
        mov   di, -1                    ; Memorare semn '-'
atoi_3:
        inc   si
atoi_4:
        xor   ax, ax                    ; Valoare initiala
        xor   ch, ch                    ; Pentru adunare cifra
        mov   bx, 10                    ; Deinmultit
atoi_5:
        cmp   byte ptr [si], '0'        ; Test
        jb    atoi_6                    ; cifre
        cmp   byte ptr [si],'9'         ; zecimale
        ja    atoi_6
        mul   bx                        ; n = n * 10
        mov   cl, [si]                  ; +
        sub   cl, '0'                   ; cifra
        add   ax, cx                    ; curenta
        inc   si
        jmp   atoi_5                    ; Reluare bucla
atoi_6:
        cmp   di, -1                    ; A fost '-' ?
        jne   atoi_7
```

```
            neg   ax                        ; Schimba semnul
      atoi_7:
            pop   dx                        ; Refaceri
            pop   cx
            pop   bx
            pop   si
            pop   di
            retf
      atoi_proc endp

      puts_proc proc far
;
;     Afiseaza sirul de la adresa DS:SI
;
            push  ax                        ; Refaceri
            push  si
            push  dx
      puts_1:
            mov   dl, [si]                  ; Preia caracter
            test  dl, dl                    ; Test terminator
            jz    puts_2
            dosint      2                   ; Afisare caracter
            inc   si                        ; Urmatorul
            jmp   puts_1
      puts_2:
            pop dx                          ; Refaceri
            pop si
            pop ax
            retf
      puts_proc endp
;
;     Buffer local pentru gets_proc
;
      .data
            buffer      db 83 dup (?)
      .code

      gets_proc proc far
;
;     Citeste un sir de lungime maxima CX si il depune
;     la adresa DS:DI
;     Se foloseste functia DOS 0AH, care vrea lungimea maxima pe
;     primul octet al buffer-ului. Ea raporteaza cate caractere
;     s-au citit de fapt in al doilea octet al buffer-ului.
;     Caracterele sunt depuse incepand de la al treilea octet al
;     buffer-ului. Lungimea raportata nu include CR final,
;     care se depune totusi in buffer-ul local. Functia 0AH
;     accepta MAX-1
;     caractere, unde MAX este primul octet din buffer, dupa care
;     se da un semnal sonor si se asteapta obligatoriu CR.

            push  ax                        ; Salvari
            push  dx
            push  di
            push  si
```

```
        push   es

        mov    ax, ds                  ; Vom folosi
                                       ; instructiuni
        mov    es, ax                  ; cu siruri
        cmp    cx, 80                  ; Trunchiem lungimea
        jle    gets_1                  ; maxima la 80 de
        mov    cx, 80                  ; caractere
gets_1:
        cmp    cx, 2                   ; si la
        jge    gets_2                  ; minim 2 caractere
        mov    cx, 2                   ; (pentru CR
                                       ; si terminator
gets_2:

        mov    buffer, cl              ; Pregatire buffer
        lea    dx, buffer
        dosint 0AH                     ; Apel functie
        mov    cl, buffer+1            ; Lungime efectiva
        mov    ch, 0                   ; Pentru JCXZ
        jcxz   gets_3                  ; Daca este 0, e
                                       ; sirul vid
        lea    si, buffer+2            ; Caracterele
                                       ; propriu-zise
        cld                            ; Directie ascendenta

        rep    movsb                   ; Bucla de copiere in
                                       ; in spatiul
                                       ; utilizatorului
gets_3:
        mov    byte ptr es:[di], 0     ; Terminator sir
        mov    dl, 0DH                 ; Un CR/LF in ecou
        dosint 2                       ; la
        mov    dl, 0AH                 ; consola
        dosint      2

        pop es                         ; Refaceri
        pop si
        pop di
        pop dx
        pop ax
        retf
gets_proc endp

getc_proc proc far
;
;       Citeste caracter în AL
;
        dosint 1
        mov    ah, 0                   ; Intoarce ca intreg
        retf
getc_proc endp

putc_proc proc far
;
```

```
;       Afiseaza caracterul din AL
;
              push  ax
              push  dx
              mov   dl, al
              dosint 2
              pop   dx
              pop   ax
              retf
       putc_proc endp

       putu_proc   proc  far
;
;      Afiseaza intreg fara semn. Parametrul in stiva.
;
              push  bp
              mov   bp, sp
              push  dx                        ; Refaceri
              push  ax
              push  bx

              mov   ax, [bp+6]                ; Preia intreg din
                                              ; stiva
              cmp   ax, 10
              mov   dl, al
              jb    putu_1                    ; Este mai mic decat 10
              mov   bx, 10                    ; Nu, il impartim
              xor   dx, dx                    ; 32 de biti la 16
              div   bx                        ; AX = cat (n/10)
                                              ; DX = rest (n MOD 10)
              push  ax                        ; Apel
              call  far ptr putu_proc         ; recursiv pentru
              add   sp, 2                     ; n/10
       putu_1:
              add   dl, '0'                   ; Afisare n MOD 10
              dosint 2

              pop   bx                        ; Refaceri
              pop   ax
              pop   dx
              pop   bp
              retf
       putu_proc   endp

       puti_proc   proc  far
;
;      Afiseaza intreg cu semn. Parametrul in stiva
;
              push  bp
              mov   bp, sp

              push  ax                        ; Salvari
              push  dx

              mov   ax, [bp+6]                ; Preia numar
```

```
            or      ax, ax                  ; Test semn
            jns     puti_1                  ; E pozitiv, salt
            mov     dl, '-'                 ; E negativ; se
                                            ; afiseaza un '-'
            dosint 2
            neg     word ptr [bp+6]         ; Se schimba semnul
                                            ; numarului
      puti_1:
            push    [bp+6]                  ; Se afiseaza ca numar
                                            ; fara semn
            call    far ptr putu_proc
            add     sp, 2                   ; Descarcare stiva

            pop     dx                      ; Refaceri
            pop     ax
            pop     bp
            retf
      puti_proc     endp
end
```

Bibliografie

1. Y.-C. Liu, G.A. Gibson, "**Microcomputer Systems: The 8086/8088 Family. Arhitecture, Programming and Design**", *Prentice Hall Inc.*, Englewood Cliffs, N.J., 1984
2. "**MCS-86 Macro Assembly Language Reference Manual**", *Intel Corporation*, Santa Clara, California, 1979
3. I. Athanasiu, A. Pănoiu, "**Microprocesoarele 8086, 286, 386**", *Editura Teora*, București, 1992
4. R.C. Detmer, "**Fundamentals of Assembly Language Programming**", *D.C. Heath and Comp.*, Lexington, Mașachusetts, 1990
5. P. Abel, "**IBM PC Assembly Language and Programming**", Second Edition, *Prentice-Hall International Inc.*, Englewood Cliffs, N.J., 1991
6. F. Munteanu, G. Muscă, F. Moraru, "**C-Tehnici de Programare**", *Editura Joint Printing House*, București, 1995

Teora – Cartea prin poștă

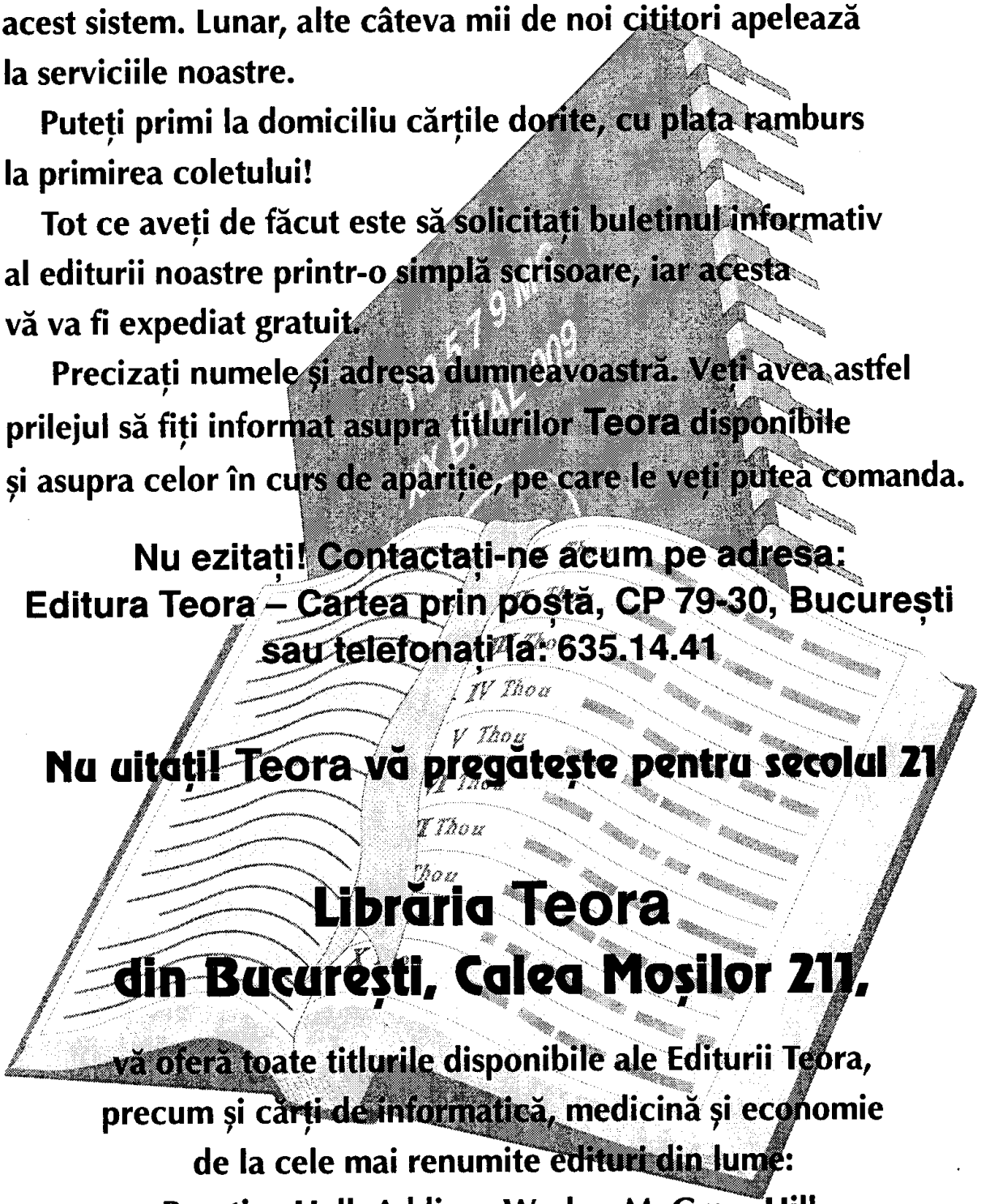
Peste 90.000 cititori (septembrie 1997) beneficiază deja de acest sistem. Lunar, alte câteva mii de noi cititori apelează la serviciile noastre.

Puteți primi la domiciliu cărțile dorite, cu plata ramburs la primirea coletului!

Tot ce aveți de făcut este să solicitați buletinul informativ al editurii noastre printr-o simplă scrisoare, iar acesta vă va fi expediat gratuit.

Precizați numele și adresa dumneavoastră. Veți avea astfel prilejul să fiți informat asupra titlurilor Teora disponibile și asupra celor în curs de apariție, pe care le veți putea comanda.

Nu ezitați! Contactați-ne acum pe adresa:
Editura Teora – Cartea prin poștă, CP 79-30, București
sau telefonați la: 635.14.41



Nu uitați! Teora vă pregătește pentru secolul 21

Librăria Teora
din București, Calea Moșilor 21,

vă oferă toate titlurile disponibile ale Editurii Teora, precum și cărți de informatică, medicină și economie de la cele mai renumite edituri din lume:
Prentice Hall, Addison Wesley, McGraw-Hill,
Harcourt Brace, John Wiley

Vizitați-ne în universul fascinant al cărții!

Limbaje și tehnici noi de programare

PROGRAMARE ÎN LIMBAJ DE ASAMBLARE

- O descriere a limbajului de asamblare (ASM) pentru procesoarele din familia Intel, având ca mediu de dezvoltare produsele Borland.
- O dezvoltare de aplicații mixte, în limbaje de nivel înalt și în ASM.
- Mijloace de dezvoltare a unor module ASM eficiente și integrarea lor în aplicații complexe.

Arhitectura 8086.
Setul de instrucțiuni 8086.
Procesoarele de 32 de biți și coprocesoarele matematice.
Structura programelor.
Macroinstrucțiuni.
Tehnici de programare în limbaj de asamblare.
Interfața ASM cu limbaje de nivel înalt.
Interfața cu sistemul de operare DOS.

ISBN 973-601-480-0
9 789736 014802

Editura Teora
Lei 20.000

❖ ❖ ❖

Limbajul de asamblare permite înțelegerea la nivel de amănunt a ceea ce se întâmplă în realitate într-un calculator. El oferă viteze de execuție ridicate și acces la hardware, care nu pot fi disponibile, cel mai adesea, în limbajele de nivel înalt.

De asemenea, aplicațiile dezvoltate în limbaje de nivel înalt devin mai performante dacă anumite module critice sunt scrise în limbaj de asamblare.

Lucrarea de față îi înlesnește cititorului o înțelegere profundă a posibilităților hardware ale calculatorului și a resurselor software oferite de mediile de dezvoltare actuale.

❖ ❖ ❖

Despre autor:

Gheorghe MUSCĂ a absolvit *Facultatea de Automatică și Calculatoare* din Universitatea „POLITEHNICA" București în anul 1981 și *Facultatea de Matematică* din Universitatea București în anul 1990.

În prezent este conferențiar la *Catedra de Calculatoare de Proces și Automatizări* din cadrul *Facultății de Automatică și Calculatoare*, unde predă cursuri de *Programare în limbaj de asamblare, Tehnici de programare* și *Proiectarea și programarea sistemelor în timp real.* Este membru al *Societății Române de Automatică și Informatică Tehnică.* A publicat 6 cărți și manuale, precum și 35 de articole de specialitate. Domeniile de interes profesional includ sistemele în timp real și proiectarea aplicațiilor de conducere distribuită a proceselor.

Este doctor inginer în specialitatea „Sisteme automate".

G. Berceanu
Iași 08.12.'97.

GHEORGHE MUSCĂ

Programare în limbaj de asamblare

Limbaje și tehnici noi de programare

Teora

Titlul: Programare în limbaj de asamblare

Copyright ©1997 **Teora**
Toate drepturile asupra acestei cărți aparțin Editurii **Teora**
Reproducerea integrală sau parțială a textului din această carte este posibilă numai cu acordul prealabil scris al Editurii **Teora.**

Distribuție

București: B-dul Al. I. Cuza nr. 39; tel./fax: 222.45.33
Sibiu: Șos. Alba Iulia nr.40; tel.069/21.04.72; fax 069/23.51.27
Bacău: Calea Mărășești nr. 5; tel./fax: 034/18.18.26

Teora - Cartea prin poștă
CP 79-30, cod 72450 București, România
Tel./fax: 635.14.41

Teora
CP 79-30, cod 72450 București, România
Fax: 210.38.28

Coperta: Valentin Tănase
NOT: 2032 CAL LIMBAJ DE ASAMBL, PROGR AR
ISBN: 973-601-480-0
Printed in Romania